COURSENOTES

CS2604:
Data Structures and File Processing
C++ Edition

Clifford A. Shaffer
Department of Computer Science
Virginia Tech
The Need for Data Structures

Data structures organize data
⇒ more efficient programs.

- More powerful computers ⇒ more complex applications.
- More complex applications demand more calculations.
- Complex computing tasks are unlike our everyday experience.

Any organization for a collection of records can be searched, processed in any order, or modified.

- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.
Efficiency

A solution is said to be **efficient** if it solves the problem within its **resource constraints**.

- space
- time

The **cost** of a solution is the amount of resources that the solution consumes.
Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Some questions to ask:

- Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be deleted?
- Are all data processed in some well-defined order, or is random access allowed?
Data Structure Philosophy

Each data structure has costs and benefits.

Rarely is one data structure better than another in all situations.

A data structure requires:
• space for each data item it stores,
• time to perform each basic operation,
• programming effort.

Each problem has constraints on available space and time.

Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:
• Start account: a few minutes
• Transactions: a few seconds
• Close account: overnight
Goals of this Course

1. Reinforce the concept that there are costs and benefits for every data structure.

2. Learn the commonly used data structures. These form a programmer’s basic data structure “toolkit.”

3. Understand how to measure the effectiveness of a data structure or program.
   • These techniques also allow you to judge the merits of new data structures that you or others might invent.
Definitions

A **type** is a set of values.

A **data type** is a type and a collection of operations that manipulate the type.

A **data item** or **element** is a piece of information or a record.

A data item is said to be a **member** of a data type.

A **simple data item** contains no subparts.

An **aggregate data item** may contain several pieces of information.
Abstract Data Types

**Abstract Data Type** (ADT): a definition for a data type solely in terms of a set of values and a set of operations on that data type.

Each ADT operation is defined by its inputs and outputs.

**Encapsulation**: hide implementation details

A **data structure** is the physical implementation of an ADT.

- Each operation associated with the ADT is implemented by one or more subroutines in the implementation.

**Data structure** usually refers to an organization for data in main memory.

**File structure**: an organization for data on peripheral storage, such as a disk drive or tape.

An ADT manages complexity through abstraction: **metaphor**.
Logical vs. Physical Form

Data items have both a **logical** and a **physical** form.

Logical form: definition of the data item within an ADT.

Physical form: implementation of the data item within a data structure.
Problems

**Problem**: a task to be performed.
- Best thought of as inputs and matching outputs.
- Problem definition should include constraints on the resources that may be consumed by any acceptable solution.

Problems ⇔ mathematical functions
- A **function** is a matching between inputs (the **domain**) and outputs (the **range**).
- An **input** to a function may be single number, or a collection of information.
- The values making up an input are called the **parameters** of the function.
- A particular input must always result in the same output every time the function is computed.
Algorithm: a method or a process followed to solve a problem.

An algorithm takes the input to a problem (function) and transforms it to the output.

A problem can have many algorithms.

An algorithm possesses the following properties:
1. It must be correct.
2. It must be composed of a series of concrete steps.
3. There can be no ambiguity as to which step will be performed next.
4. It must be composed of a finite number of steps.
5. It must terminate.

A computer program is an instance, or concrete representation, for an algorithm in some programming language.
Set concepts and notation

Recursion

Induction proofs

Logarithms

Summations
Estimation Techniques

Known as “back of the envelope” or “back of the napkin” calculation.

1. Determine the major parameters that affect the problem.
2. Derive an equation that relates the parameters to the problem.
3. Select values for the parameters, and apply the equation to yield an estimated solution.

Example:
How many library bookcases does it take to store books totaling one million pages?

Estimate:
- pages/inch
- feet/shelf
- shelves/bookcase
Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code and debug.
2. To design an algorithm that makes efficient use of the computer’s resources.

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm’s cost?
How to Measure Efficiency?

1. Empirical comparison (run programs).

2. Asymptotic Algorithm Analysis.

Critical resources:

Factors affecting running time:

For most algorithms, running time depends on “size” of the input.

Running time is expressed as $T(n)$ for some function $T$ on input size $n$. 
Examples of Growth Rate

Example 1:

```c
int largest(int* array, int n) { // Find largest value
    int currlarge = array[0]; // Store largest seen
    for (int i=1; i<n; i++) // For each element
        if (array[i] > currlarge) // If largest
            currlarge = array[i]; // Remember it
    return currlarge; // Return largest
}
```

Example 2: Assignment statement

Example 3:

```c
sum = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum++;
```
Gro wth Rate Graph

Input size $n$
Best, Worst and Average Cases

Not all inputs of a given size take the same time.

Sequential search for $K$ in an array of $n$ integers:
- Begin at first element in array and look at each element in turn until $K$ is found.

Best Case:

Worst Case:

Average Case:

While average time seems to be the fairest measure, it may be difficult to determine.

When is worst case time important?
Faster Computer or Algorithm?

What happens when we buy a computer 10 times faster?

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th>$n$</th>
<th>$n'$</th>
<th>Change</th>
<th>$n'/n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10n$</td>
<td>1,000</td>
<td>10,000</td>
<td>$n'=10n$</td>
<td>10</td>
</tr>
<tr>
<td>$20n$</td>
<td>500</td>
<td>5,000</td>
<td>$n'=10n$</td>
<td>10</td>
</tr>
<tr>
<td>$5n\log n$</td>
<td>250</td>
<td>1,842</td>
<td>$\sqrt{10n}&lt;n'&lt;10n$</td>
<td>7.37</td>
</tr>
<tr>
<td>$2n^2$</td>
<td>70</td>
<td>223</td>
<td>$n'=\sqrt{10n}$</td>
<td>3.16</td>
</tr>
<tr>
<td>$2^n$</td>
<td>13</td>
<td>16</td>
<td>$n'=n+3$</td>
<td>--</td>
</tr>
</tbody>
</table>

$n$: Size of input that can be processed in one hour (10,000 steps).

$n'$: Size of input that can be processed in one hour on the new machine (100,000 steps).
Asymptotic Analysis: Big-oh

Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the set $O(f(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.

Usage: The algorithm is in $O(n^2)$ in [best, average, worst] case.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in less than $cf(n)$ steps [in best, average or worst case].

Upper Bound.

Example: if $T(n) = 3n^2$ then $T(n)$ is in $O(n^2)$.

Wish tightest upper bound:
While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.
Big-oh Example

Example 1. Finding value $X$ in an array.

$T(n) = c_s n/2$.
For all values of $n > 1$, $c_s n/2 \leq c_s n$.
Therefore, by the definition, $T(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

Example 2. $T(n) = c_1 n^2 + c_2 n$ in average case
$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ for all $n > 1$.

$T(n) \leq cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.
Therefore, $T(n)$ is in $O(n^2)$ by the definition.

Example 3: $T(n) = c$. We say this is in $O(1)$. 
Big-Omega

Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \geq cg(n)$ for all $n > n_0$.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in more than $cg(n)$ steps.

Lower Bound.

Example: $T(n) = c_1 n^2 + c_2 n$.

$c_1 n^2 + c_2 n \geq c_1 n^2$ for all $n > 1$.
$T(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$.

Therefore, $T(n)$ is in $\Omega(n^2)$ by the definition.

Want greatest lower bound.
Theta Notation

When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.

Definition: An algorithm is said to be Θ(h(n)) if it is in O(h(n)) and it is in Ω(h(n)).

Simplifying Rules:

1. If \( f(n) \) is in \( O(g(n)) \) and \( g(n) \) is in \( O(h(n)) \), then \( f(n) \) is in \( O(h(n)) \).

2. If \( f(n) \) is in \( O(kg(n)) \) for any constant \( k > 0 \), then \( f(n) \) is in \( O(g(n)) \).

3. If \( f_1(n) \) is in \( O(g_1(n)) \) and \( f_2(n) \) is in \( O(g_2(n)) \), then \( (f_1 + f_2)(n) \) is in \( O(\max(g_1(n), g_2(n))) \).

4. If \( f_1(n) \) is in \( O(g_1(n)) \) and \( f_2(n) \) is in \( O(g_2(n)) \) then \( f_1(n)f_2(n) \) is in \( O(g_1(n)g_2(n)) \).
Running Time of a Program

Example 1: \( a = b; \)

This assignment takes constant time, so it is \( \Theta(1). \)

Example 2:

\[
\text{sum} = 0; \\
\text{for (i=1; i<=n; i++)} \\
\quad \text{sum} += n;
\]

Example 3:

\[
\text{sum} = 0; \\
\text{for (j=1; j<=n; j++)} \quad \text{// First for loop} \\
\quad \text{for (i=1; i<=j; i++)} \quad \text{// is a double loop} \\
\quad \quad \text{sum}++; \\
\text{for (k=0; k<n; k++)} \quad \text{// Second for loop} \\
\quad \text{A}[k] = k;
\]
More Examples

Example 4.

```c
sum1 = 0;
for (i=1; i<=n; i++) // First double loop
    for (j=1; j<=n; j++) // do n times
        sum1++;

sum2 = 0;
for (i=1; i<=n; i++) // Second double loop
    for (j=1; j<=i; j++) // do i times
        sum2++;
```

Example 5.

```c
sum1 = 0;
for (k=1; k<=n; k*=2)
    for (j=1; j<=n; j++)
        sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)
    for (j=1; j<=k; j++)
        sum2++;
```
int binary(int K, int* array, int left, int right) {
    // Return position of element (if any) with value K
    int l = left-1;
    int r = right+1;  // l, r are beyond array bounds
    while (l+1 != r) { // Stop when l and r meet
        int i = (l+r)/2;  // Look at middle of subarray
        if (K < array[i]) r = i;  // In left half
        if (K == array[i]) return i;  // Found it
        if (K > array[i]) l = i;    // In right half
    }
    return UNSUCCESSFUL;  // Search value not in array
}

Analysis: How many elements can be examined in the worst case?
Other Control Statements

while loop: analyze like a for loop.

if statement: Take greater complexity of then/else clauses.

switch statement: Take complexity of most expensive case.

Subroutine call: Complexity of the subroutine.
Analyzing Problems

Upper bound: Upper bound of best known algorithm.

Lower bound: Lower bound for every possible algorithm.

Example: Sorting
1. Cost of I/O: $\Omega(n)$
2. Bubble or insertion sort: $O(n^2)$
3. A better sort (Quicksort, Mergesort, Heapsort, etc.): $O(n \log n)$
4. We prove later that sorting is $\Omega(n \log n)$
Multiple Parameters

Compute the rank ordering for all $C$ pixel values in a picture of $P$ pixels.

```c
for (i=0; i<C; i++) // Initialize count
    count[i] = 0;
for (i=0; i<P; i++) // Look at all of the pixels
    count[value(i)]++; // Increment proper value count
sort(count); // Sort pixel value counts
```

If we use $P$ as the measure, then time is $\Theta(P \log P)$.

More accurate is $\Theta(P + C \log C)$.
Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm
Space: Data Structure

Space/Time Tradeoff Principle:
One can often achieve a reduction in time if one is willing to sacrifice space, or vice versa.

- Encoding or packing information
  - Boolean flags
- Table lookup
  - Factorials

Disk Based Space/Time Tradeoff Principle:
The smaller you can make your disk storage requirements, the faster your program will run.
Lists

A list is a finite, ordered sequence of data items called elements.

Each list element has a data type.

The empty list contains no elements.

The length of the list is the number of elements currently stored.

The beginning of the list is called the head, the end of the list is called the tail.

Sorted lists have their elements positioned in ascending order of value, while unsorted lists have no necessary relationship between element values and positions.

Notation: (a_0, a_1, ..., a_{n-1})

What operations should we implement?
class List { // List class ADT
public:
    List(int =LIST_SIZE); // Constructor
    ~List(); // Destructor
    void clear(); // Remove all Elems
    void insert(const Elem); // Insert Elem at curr
    void append(const Elem); // Insert Elem at tail
    Elem remove(); // Remove and return Elem
    void setFirst(); // Set curr to first pos
    void prev(); // Move curr to prev pos
    void next(); // Move curr to next pos
    int length() const; // Return current length
    void setPos(int); // Set curr to position
    void setValue(const Elem); // Set current value
    Elem currValue() const; // Return current value
    bool isEmpty() const; // TRUE if list is empty
    bool isInList() const; // TRUE if curr in list
    bool find(int); // Find value
};
List ADT Examples

List: (12, 32, 15)

MyList.insert(99);

Assume MyPos has 32 as current element:

Process an entire list:

```java
for (MyList.first(); MyList.isInList(); MyList.next())
    DoSomething(MyList.currValue());
```
Array-Based List Insert

Insert 23:

(a)  

(b)  

(c)
Array-Based List Class

class List { // Array-based list class
    private:
        int msize; // Maximum size of list
        int numinlist; // Actual number of Elems
        int curr; // Position of "current"
        Elem* listarray; // Array of list Elems
    public:
        List(int =LIST_SIZE); // Constructor
        ~List(); // Destructor
        void clear(); // Remove all Elems
        void insert(const Elem); // Insert Elem at curr
        void append(const Elem); // Insert Elem at tail
        Elem remove(); // Remove and return Elem
        void setFirst(); // Set curr to first pos
        void prev(); // Move curr to prev pos
        void next(); // Move curr to next pos
        int length() const; // Return current length
        void setPos(int); // Set curr to position
        void setValue(const Elem); // Set current value
        Elem currValue() const; // Return current value
        bool isEmpty() const; // TRUE if list is empty
        bool isInList() const; // TRUE if curr in list
        bool find(int); // Find value
};
Array-Based List Implementation

List::List(int sz) // Constructor
{ msize = sz; numinlist = 0; curr = 0;
  listarray = new Elem[msize]; }

List::~List() { delete [] listarray; } // Destructor

void List::clear() { numinlist = 0; curr = 0; }

// Insert Elem at current position
void List::insert(const Elem item) {
  assert((numinlist < msize) && (curr >=0) &&
  (curr <= numinlist));
  for(int i=numinlist; i>curr; i--) // Shift Elems up
    listarray[i] = listarray[i-1];
  listarray[curr] = item;
  numinlist++; // Increment size
}

void List::append(const Elem item) { // Insert at tail
  assert(numinlist < msize); // Can’t be full
  listarray[numinlist++] = item; // Increment size
}

Elem List::remove() { // Remove/return current Elem
  assert(!isEmpty() && isInList()); // Must have Elem
  Elem temp = listarray[curr]; // Store Elem
  for(int i=curr; i<numinlist-1; i++) // Shift down
    listarray[i] = listarray[i+1];
  numinlist--; return temp;
}
void List::setFirst() { curr = 0; } // Set to first pos
void List::prev() { curr--; } // Move curr to prev pos
void List::next() { curr++; } // Move curr to next pos
int List::length() const { return numinlist; }
void List::setPos(int pos) { curr = pos; }
void List::setValue(const Elem val) // Set curr value
{ assert(isInList()); listarray[curr] = val; }
Elem List::currValue() const // Return current value
{ assert(isInList()); return listarray[curr]; }
bool List::isEmpty() const { return numinlist == 0; }
bool List::isInList() const // TRUE if in list
{ return (curr >= 0) && (curr < numinlist); }
bool List::find(int val) { // Find value
while (isInList()) // Stop if reach end
if (key(currValue()) == val) return TRUE; // Found
else next();
return FALSE; // Not found
}
Link Class

Dynamic allocation of new list elements.

class Link {
    // Singly-linked node
public:
    Elem element;  // Elem value for node
    Link *next;    // Pointer to next node
    Link(const Elem elemval, Link* nextval = NULL) {
        element = elemval;  next = nextval; }
    Link(Link* nextval = NULL) { next = nextval; }
};
Linked List Position

(a) Naive approach: Point to current node. Current is 12. Want to insert node with 10. No access available to node with 12. How can we do the insert?

(b) Alternate implementation: Point to node preceding actual current node. No we can do the insert. Also note use of header node.
class List { // Linked list class
    private:
        Link* head; // Pointer to list header
        Link* tail; // Pointer to last Elem
        Link* curr; // Pos of "current" Elem
    public:
        List(int =LIST_SIZE); // Constructor
        ~List(); // Destructor
        void clear(); // Remove all Elems
        void insert(const Elem); // Insert at current pos
        void append(const Elem); // Insert at tail of list
        Elem remove(); // Remove/return Elem
        void setFirst(); // Set curr to first pos
        void prev(); // Move curr to prev pos
        void next(); // Move curr to next pos
        int length() const; // Return length
        void setPos(int); // Set current pos
        void setValue(const Elem); // Set current value
        Elem currValue() const; // Return current value
        bool isEmpty() const; // TRUE if list is empty
        bool isInList() const; // TRUE if now in list
        bool find(int); // Find value
};
// Insert Elem at current position
void List::insert(const Elem item) {
    assert(curr != NULL); // Must be pointing to Elem
    curr->next = new Link(item, curr->next);
    if (tail == curr) // Appended new Elem
        tail = curr->next;
}
Elem List::remove() { // Remove/return Elem
    assert(isInList()); // Must be valid pos
    Elem temp = curr->next->element; // Remember value
    Link* ltemp = curr->next; // Remember link
    curr->next = ltemp->next; // Remove from list
    if (tail == ltemp) tail = curr; // Set tail
    delete ltemp; // Free link
    return temp; // Return value
}
Freelists

System new and delete are slow.

class Link { // Singly-linked node
public: // with freelist
    Elem element; // Elem value for node
    Link* next; // Pointer to next node
static Link* freelist; // Link class freelist
Link(const Elem elemval, Link* nextval = NULL)
    { element = elemval; next = nextval; }
Link(Link* nextval = NULL) { next = nextval; }
void* operator new(size_t); // Overloaded new
void operator delete(void*); // Overloaded delete
};

Link* Link::freelist = NULL; // Create static variable

void* Link::operator new(size_t) { // Overload new
    if (freelist == NULL) return ::new Link; // New space
    Link* temp = freelist; // Or get from freelist
    freelist = freelist->next; // Return the link
    return temp;
}

void Link::operator delete(void* ptr) { // Overload
    ((Link*)ptr)->next = freelist; // Put on freelist
    freelist = (Link*)ptr;
}
Comparison of List Implementations

Array-Based Lists:
- Insertion and deletion are $\Theta(n)$.
- Array must be allocated in advance.
- No overhead if all array positions are full.

Linked Lists:
- Insertion and deletion $\Theta(1)$;
  prev and direct access are $\Theta(n)$.
- Space grows with number of elements.
- Every element requires overhead.

Space “break-even” point:

$$DE = n(P + E); \quad n = \frac{DE}{P + E}$$

E: Space for data value
P: Space for pointer
D: Number of elements in array
Doubly Linked Lists

Simplify insertion and deletion: Add a prev pointer.

```cpp
class Link { // Doubly-linked node
    public: // with freelist
        Elem element; // Node Elem value
        Link* next; // Pointer to next node
        Link* prev; // Pointer to prev node
        static Link* freelist; // Link class freelist
        Link(const Elem Elemval, Link* nextp = NULL,
             Link* prevp = NULL)
            { element = Elemval; next = nextp; prev = prevp; }
        Link(Link* nextp = NULL, Link* prevp = NULL)
            { next = nextp; prev = prevp; }
        void* operator new(size_t); // Overloaded new
        void operator delete(void*); // Overloaded delete
};
```

![Diagram of doubly linked list with head, curr, and tail nodes and values 20, 23, 12, 15.]
Doubly Linked List Operations

// Insert Elem at current position
void List::insert(const Elem item) {
    assert(curr != NULL);
    curr->next = new Link(item, curr->next, curr);
    if (curr->next->next != NULL)
        curr->next->next->prev = curr->next;
    if (tail == curr) tail = curr->next;
}

Elem List::remove() { // Remove current Elem
    assert(isInList()); // Must be valid position
    Elem temp = curr->next->element;
    Link* ltemp = curr->next;
    if (ltemp->next != NULL) ltemp->next->prev = curr;
    else tail = curr; // Removed tail Elem - change tail
    curr->next = ltemp->next;
    delete ltemp;
    return temp;
}
Stacks

LIFO: Last In, First Out

Restricted form of list: Insert and remove only at front of list.

Notation:
- Insert: PUSH
- Remove: POP
- The accessible element is called TOP.
Array-Based Stack

Define top as first free position.

class Stack { // Array-based stack class
private:
    int size;  // Maximum size of stack
    int top;   // Index for top Elem
    Elem *listarray; // Array holding stack Elems

public:
    Stack(int sz =LIST_SIZE) // Constructor: initialize
        { size = sz; top = 0; listarray = new Elem[sz]; }
    ~Stack() // Destructor: free array
        { delete [] listarray; }
    void clear() // Remove all Elems
        { top = 0; }
    void push(const Elem item) // Push Elem onto stack
        { assert(top < size); listarray[top++] = item; }
    Elem pop() // Pop Elem from stack top
        { assert(!isEmpty()); return listarray[--top]; }
    Elem topValue() const // Return value of top Elem
        { assert(!isEmpty()); return listarray[top-1]; }
    bool isEmpty() const // Return TRUE if empty
        { return top == 0; }
};
class Stack { // Linked stack class
private:
    Link *top; // Pointer to top Elem
public:
    Stack(int sz = LIST_SIZE) // Constructor:
        { top = NULL; } // initialize
    ~Stack() { clear(); } // Destructor
    void clear(); // Remove stack Elems
    void push(const Elem item) // Push Elem onto stack
        { top = new Link(item, top); }
    Elem pop(); // Pop Elem from stack
    Elem topValue() const // Get value of top Elem
        { assert(!isEmpty()); return top->element; }
    bool isEmpty() const // Return TRUE if empty
        { return top == NULL; }
};

void Stack::clear() { // Remove Elems
    while(top != NULL) // Free link nodes
        { Link* temp = top; top = top->next; delete temp; }
}

Elem Stack::pop() { // Pop Elem from stack
    assert(!isEmpty());
    Elem temp = top->element;
    Link* ltemp = top->next;
    delete top; top = ltemp;
    return temp;
}
Queues

FIFO: First In, First Out

Restricted form of list:
   Insert at one end, remove from other.

Notation:
   • Insert: Enqueue
   • Delete: Dequeue
   • First element: FRONT
   • Last element: REAR
Queue Implementations

Array-Based Queue

![Array-Based Queue Diagram](image)

Linked Queue: modified linked list.
Binary Trees

A binary tree is made up of a finite set of nodes that is either empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root.

Notation: Node, Children, Edge, Parent, Ancestor, Descendant, Path, Depth, Height, Level, Leaf Node, Internal Node, Subtree.
Full and Complete Binary Trees

**Full** binary tree: each node either is a leaf or is an internal node with exactly two non-empty children.

**Complete** binary tree: If the height of the tree is $d$, then all levels except possibly level $d$ are completely full. The bottom level has all nodes to the left side.

![Diagram](a)

![Diagram](b)
**Full Binary Tree Theorem**

Theorem: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

Proof (by Mathematical Induction):

- **Base Case**: A full binary tree with 1 internal node must have two leaf nodes.

- **Induction Hypothesis**: Assume any full binary tree $T$ containing $n - 1$ internal nodes has $n$ leaves.

- **Induction Step**: Given tree $T$ with $n$ internal nodes, pick internal node $I$ with two leaf children. Remove $I$’s children, call resulting tree $T'$. By induction hypothesis, $T'$ is a full binary tree with $n$ leaves. Restore $i$’s two children. The number of internal nodes has now gone up by 1 to reach $n$. The number of leaves has also gone up by 1.
Full Binary Tree Theorem Corollary

**Theorem:** The number of NULL pointers in a non-empty binary tree is one more than the number of nodes in the tree.

**Proof:** Replace all null pointers with a pointer to an empty leaf node. This is a full binary tree.
Binary Tree Node Class

class BinNode { // Binary tree node class
    public:
    Belem element; // The node’s value
    BinNode* left; // Pointer to left child
    BinNode* right; // Pointer to right child
    static BinNode* freelist;
    // Two constructors: with and without initial values
    BinNode() { left = right = NULL; }
    BinNode(Belem e, BinNode* l = NULL, BinNode* r = NULL)
        { element = e; left = l; right = r; }
    ~BinNode() { } // Destructor
    BinNode* leftchild() const { return left; }
    BinNode* rightchild() const { return right; }
    Belem value() const { return element; }
    void setValue(Belem val) { element = val; }
    bool isLeaf() const // TRUE if is a leaf
        { return (left == NULL) && (right == NULL); }
    void* operator new(size_t); // Overload new
    void operator delete(void*); // Overload delete
};
Traversals

Any process for visiting the nodes in some order is called a traversal.

Any traversal that lists every node in the tree exactly once is called an enumeration of the tree’s nodes.

Preorder traversal: Visit each node before visiting its children.

Postorder traversal: Visit each node after visiting its children.

Inorder traversal: Visit the left subtree, then the node, then the right subtree.

```c
void preorder(BinNode* rt) // rt is root of a subtree
{
    if (rt == NULL) return; // Empty subtree
    visit(rt); // visit performs desired action
    preorder(rt->leftchild());
    preorder(rt->rightchild());
}
```
Binary Tree Implementation

Example of expression tree:

```
(4 * 2) + a - c
```

Leaves are different from internal nodes.
enum Nodetype {leaf, internal}; // Enumerate node types
class VarBinNode { // Generic node class
public:
    Nodetype mytype; // Stores type for this node
    union {
        struct { // Structure for internal node
            VarBinNode* left; VarBinNode* right; // Children
            Operator opx; // Internal node value
        } intl;
        Operand var; // Leaves just store a value
    };
    VarBinNode(const Operand& val) // Constructor: leaf
        { mytype = leaf; var = val; }
    // Constructor: Internal
    VarBinNode(const Operator& op,
                VarBinNode* l, VarBinNode* r) {
        mytype = internal; intl.opx = op;
        intl.left = l; intl.right = r;
    }
    bool isLeaf() { return mytype == leaf; }
    VarBinNode* leftchild() { return intl.left; }
    VarBinNode* rightchild() { return intl.right; }
};

void traverse(VarBinNode* rt) { // Preorder traversal
    if (rt == NULL) return;
    if (rt->isLeaf()) cout << "Leaf: " << rt->var << "\n";
    else {
        cout << "Internal: " << rt->intl.opx << "\n";
        traverse(rt->leftchild());
        traverse(rt->rightchild());
    }
}
Inheritance

class VarBinNode {  // Node base class
public:
    // isLeaf is a "pure" virtual function.
    // Each derived class must define isLeaf.
    virtual bool isLeaf() = 0;
};

class LeafNode : public VarBinNode {  // Leaf subclass
private:
    Operand var;  // Operand value
public:
    LeafNode(const Operand& val) { var = val; }  // Constructor
    bool isLeaf() { return TRUE; }  // Subclass version
    Operand value() { return var; }  // Return value
};

class IntlNode : public VarBinNode {  // Internal node
private:
    VarBinNode* left;  // Left child
    VarBinNode* right;  // Right child
    Operator opx;  // Operator value
public:
    IntlNode(const Operator& op,
             VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; }  // Constructor
    bool isLeaf() { return FALSE; }  // Subclass version
    VarBinNode* leftch() { return left; }  // Left child
    VarBinNode* rightch() { return right; }  // Right child
    Operator value() { return opx; }  // Value
};
Inheritance (cont)

void traverse(VarBinNode *rt) {  // Preorder traversal
  if (rt == NULL) return;  // Nothing to visit
  if (rt->isLeaf()) {  // Do leaf node
    cout << "Leaf: " << ((LeafNode *)rt)->value() << "\n";
  } else {  // Do internal node
    cout << "Internal: " << ((IntlNode *)rt)->value() << "\n";
    traverse(((IntlNode *)rt)->leftch());
    traverse(((IntlNode *)rt)->rightch());
  }
}
Space Overhead

From Full Binary Tree Theorem:
    Half of pointers are NULL.

If leaves only store information, then overhead depends on whether tree is full.

All nodes the same, with two pointers to children:
    Total space required is \((2p + d)n\).
    Overhead: \(2pn\).

If \(p = d\), this means \(2p/(2p + d) = 2/3\) overhead.

Eliminate pointers from leaf nodes:

\[
\frac{\frac{n}{2}(2p)}{\frac{n}{2}(2p) + dn} = \frac{p}{p + d}
\]

This is 1/2 if \(p = d\).

\(2p/(2p + d)\) if data only at leaves \(\Rightarrow 2/3\) overhead.

Some method is needed to distinguish leaves from internal nodes.
Array Implementation

For complete binary trees.

- Parent($r$) =
- Leftchild($r$) =
- Rightchild($r$) =
- Leftsibling($r$) =
- Rightsibling($r$) =

Since the complete binary tree is so limited in its shape, (only one shape for tree of $n$ nodes), it is reasonable to expect that space efficiency can be achieved.
Huffman Coding Trees

ASCII Codes: 8 bits per character.
  Fixed length coding.

Can take advantage of relative frequency of letters to save space.
  Variable length coding.

<table>
<thead>
<tr>
<th>Z</th>
<th>K</th>
<th>F</th>
<th>C</th>
<th>U</th>
<th>D</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>24</td>
<td>32</td>
<td>37</td>
<td>42</td>
<td>42</td>
<td>120</td>
</tr>
</tbody>
</table>

Build the tree with **minimal external path weight**.
Huffman Tree Construction

Step 1:

```
| 2 Z | 7 K | 24 F | 32 C | 37 U | 42 D | 42 L | 120 E |
```

Step 2:

```
2 Z 7 K
```

Step 3:

```
Z K
```

Step 4:

```
Z K
```

Step 5:

```
Z K
```

64
Assigning Codes

<table>
<thead>
<tr>
<th>Letter</th>
<th>Freq</th>
<th>Code</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Coding and Decoding

A set of codes are said to meet the **prefix property** if no code in the set is the prefix of another.

Code for DEED:

Decode 101100111011101:

Expected cost per letter:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Freq</th>
<th>Code</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>32</td>
<td>1110</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>42</td>
<td>101</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>120</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>24</td>
<td>11111</td>
<td>5</td>
</tr>
<tr>
<td>K</td>
<td>7</td>
<td>111101</td>
<td>6</td>
</tr>
<tr>
<td>L</td>
<td>42</td>
<td>110</td>
<td>3</td>
</tr>
<tr>
<td>U</td>
<td>37</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>Z</td>
<td>2</td>
<td>111100</td>
<td>6</td>
</tr>
</tbody>
</table>
Binary Search Trees

Binary Search Tree (BST) Property

All elements stored in the left subtree of a node whose value is $K$ have values less than $K$. All elements stored in the right subtree of a node whose value is $K$ have values greater than or equal to $K$. 

(a)  

(b)
BST Search

class BST {
private:
    BinNode* root;
    void clearhelp(BinNode*); // Private
    void inserthelp(BinNode*&, const Belem); // functions
    BinNode* deletemin(BinNode*&);
    void removehelp(BinNode*&, int);
    Belem findhelp(BinNode*, int) const;
    void printhelp(const BinNode*, int) const;

public:
    BST() { root = NULL; }
    ~BST() { clearhelp(root); }
    void clear() { clearhelp(root); root = NULL; }
    void insert(const Belem val) {inserthelp(root, val);}
    void remove(const val) { removehelp(root, val); }
    Belem find(const val) const
    { return findhelp(root, val); }
    bool isEmpty() const { return root == NULL; }
    void print() const {
        if (root == NULL) cout << "The BST is empty.\n";
        else printhelp(root, 0);
    }
};

Belem BST::findhelp(BinNode* rt, int val) const {
    if (rt == NULL) return UNSUCCESSFUL; // Empty tree
    else if (val < key(rt->value())) // Left
        return findhelp(rt->leftchild(), val);
    else if (val == key(rt->value())) return rt->value();
    else return findhelp(rt->rightchild(), val); // Right
}
void BST::insertHelp(BinNode*& rt, const Belem val) {
    if (rt == NULL) { // Empty tree: create node
        rt = new BinNode(val, NULL, NULL);
    } else if (key(val) < key(rt->value()))
        insertHelp(rt->left, val); // Check left
    else insertHelp(rt->right, val); // Check right
}

Note that rt is declared “by reference.”
Alternate Approach

```cpp
void BST::insertHelp(BinNode* rt, const Belem val) {
    if (rt == NULL)
        return new BinNode(val, NULL, NULL);
    if (key(val) < key(rt->value()))
        rt->left = insertHelp(rt->left, val);
    else rt->right = insertHelp(rt->right, val);
    return rt;
}
```
Remove Minimum Value

BinNode* BST::deletemin(BinNode*& rt) {
    assert(rt != NULL);  // Must be a node to delete
    if (rt->left != NULL)  // Continue left
        return deletemin(rt->left);
    else  // Found it
        { BinNode* temp = rt; rt = rt->right; return temp; }
}

```

```

```

```

```

```

```

```
void BST::removehelp(BinNode*& rt, int val) {
    if (rt == NULL) cout << val << " is not in tree.\n";
    else if (val < key(rt->value())) // Check left
        removehelp(rt->left, val);
    else if (val > key(rt->value())) // Check right
        removehelp(rt->right, val);
    else { // Found it: remove
        BinNode* temp = rt;
        if (rt->left == NULL) // Only a right -
            rt = rt->right; // point to right
        else if (rt->right == NULL) // Only a left -
            rt = rt->left; // point to left
        else { // Both non-empty
            temp = deletemin(rt->right); // Replace with min
            rt->setValue(temp->value()); // in right subtree
        }
        delete temp; // Free up space
    }
}

BST Remove

```
    37
   / \  
  24  42
 /   /  
7   32 40
     /   
     40   42
    /     /  
   2     120
```
Cost of BST Operations

Find:

Insert:

Remove:
Heaps

Heap: Complete binary tree with the **Heap Property:**

- Min-heap: all values less than child values.
- Max-heap: all values greater than child values.

The values in a heap are *partially ordered.*

Heap representation: normally the array based complete binary tree representation.
Building the Heap

(a) requires exchanges (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).

(b) requires exchanges (5-2), (7-3), (7-1), (6-1).
Heap ADT

class heap { // Max-heap class
    private:
        Elem* Heap; // Pointer to heap array
        int size; // Maximum size of heap
        int n; // Number of ELEMs in heap
        void siftdown(int); // Put ELEM in place
    public:
        heap(Elem*, int, int); // Constructor
        int heapsize() const; // Return current size
        bool isLeaf(int) const; // TRUE if pos is a leaf
        int leftchild(int) const; // Return L child position
        int rightchild(int) const; // Return R child position
        int parent(int) const; // Return parent position
        void insert(const Elem); // Insert value into heap
        Elem removemax(); // Remove maximum value
        Elem remove(int); // Remove specified value
        void buildheap(); // Heapify contents
};
Siftdown

For fast heap construction:

- Work from high end of array to low end.
- Call siftdown for each item.
- Don’t need to call siftdown on leaf nodes.

```cpp
void heap::buildheap() // Heapify contents
{ for (int i=n/2-1; i>=0; i--) siftdown(i); }
```

```cpp
void heap::siftdown(int pos) { // Put ELEM in place
    assert((pos >= 0) && (pos < n));
    while (!isLeaf(pos)) {
        int j = leftchild(pos);
        if ((j<(n-1)) && (key(Heap[j]) < key(Heap[j+1])))
            j++; // j now index of child with greater value
        if (key(Heap[pos]) >= key(Heap[j])) return; // Done
        swap(Heap, pos, j);
        pos = j; // Move down
    }
}
```

Cost for heap construction:

\[
\sum_{i=1}^{\log n} (i - 1) \frac{n}{2^i} \approx n.
\]
Priority Queues

A priority queue stores objects, and on request releases the object with greatest value.

Example: Scheduling jobs in a multi-tasking operating system.

The priority of a job may change, requiring some reordering of the jobs.

Implementation: use a heap to store the priority queue.

To support priority reordering, delete and re-insert. Need to know index for the object.

// Remove value at specified position
Elem heap::remove(int pos) {
    assert((pos > 0) && (pos < n));
    swap(Heap, pos, --n); // Swap with last value
    while (key(Heap[pos]) > key(Heap[parent(pos)]))
        swap(Heap, pos, parent(pos)); // Push up if large
    if (n != 0) siftdown(pos); // Push down if small
    return Heap[n];
}
General Trees

A **tree** $T$ is a finite set of one or more nodes such that there is one designated node $r$ called the root of $T$, and the remaining nodes in $(T - \{r\})$ are partitioned into $n \geq 0$ disjoint subsets $T_1, T_2, ..., T_k$, each of which is a tree, and whose roots $r_1, r_2, ..., r_k$, respectively, are children of $r$. 

[Diagram of a tree with labels for root, ancestors, parents, siblings, children, and subtrees.]
General Tree ADT

class GTNode {
public:
    GTNode(const Elem);      // Constructor
    ~GTNode();              // Destructor
    Elem value();           // Return node’s value
    bool isLeaf();          // TRUE if is a leaf
    GTNode* parent();       // Return parent
    GTNode* leftmost_child(); // Return first child
    GTNode* right_sibling(); // Return right sibling
    void setValue(Elem);    // Set node’s value
    void insert_first(GTNode* n); // Insert first child
    void insert_next(GTNode* n); // Insert right sibling
    void remove_first();     // Remove first child
    void remove_next();      // Remove right sibling
};

class GenTree {
public:
    GenTree();                // Constructor
    ~GenTree();              // Destructor
    void clear();            // Free nodes
    GTNode* root();          // Return root
    void newroot(Elem, GTNode*, GTNode*); // Combine
};
void print(GTNode* rt) { // Preorder traverse from root
    if (rt->isLeaf()) cout << "Leaf: ";
    else cout << "Internal: ";
    cout << rt->value() << "\n"; // Print or take action
    GTNode* temp = rt->leftmost_child();
    while (temp != NULL)
        { print(temp); temp = temp->right_sibling(); } }
Parent Pointer Implementation

Parent’s Index | 0 | 0 | 1 | 1 | 1 | 2 | 7 | 7 | 7
---|---|---|---|---|---|---|---|---|---
Label | R | A | B | C | D | E | F | W | X | Y | Z

Are two elements in the same tree?

```c++
bool Gentree::differ(int a, int b) {
    // Are nodes a and b in different trees?
    GTNode* root1 = FIND(&array[a]); // Find a’s root
    GTNode* root2 = FIND(&array[b]); // Find b’s root
    return root1 != root2; // Compare roots
}
```
Equivalence Classes

When joining equivalence classes, want to keep depth small.

Weighted Union Rule: join the tree with fewer nodes to the tree with more nodes.

Limits depth to $\log n$ for $n$ nodes.

Path Compression: Make all nodes visited point to root.

```cpp
class GTNode {
    // General tree node
    public:
        GTNode* par; // Parent pointer
        GTNode() { par = NULL; } // Constructor
        GTNode* parent() { return par; } // Return parent
};

class Gentree {
    // General tree: UNION/FIND
    private:
        GTNode* array; // Node array
        int size; // Size of node array
        GTNode* FIND(GTNode*) const; // Find root
    public:
        Gentree(int); // Constructor
        ~Gentree(); // Destructor
        void UNION(int, int); // Merge equivalences
        bool differ(int, int); // TRUE if not in same tree
};
```
Gentree::Gentree(int sz) {  // Constructor
    size = sz;
    array = new GTNode[sz];  // Create node array
}

Gentree::~Gentree() {  // Destructor
    delete [] array; // Free node array
}

bool Gentree::differ(int a, int b) {
    // Are nodes a and b in different trees?
    GTNode* root1 = FIND(&array[a]); // Find a’s root
    GTNode* root2 = FIND(&array[b]); // Find b’s root
    return root1 != root2; // Compare roots
}

void Gentree::UNION(int a, int b) {  // Merge subtrees
    GTNode* root1 = FIND(&array[a]); // Find a’s root
    GTNode* root2 = FIND(&array[b]); // Find b’s root
    if (root1 != root2) root2->par = root1; // Merge
Equivalence Processing Example

(a)

(b)

(c)

(d)
Path Compression Example

GTNode* Gentree::FIND(GTNode* curr) const {
    while (curr->parent() != NULL) curr = curr->par;
    return curr; // At root
}

<table>
<thead>
<tr>
<th>5</th>
<th>0</th>
<th>0</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>0</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Lists of Children

<table>
<thead>
<tr>
<th>Index</th>
<th>Val</th>
<th>Par</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
0 → 1 → 3 → 2 → 4 → 6
```

```
0 → 5
```
Leftmost Child/Right Sibling

- Two trees share the same array.
- The diagram shows a tree with nodes labeled A, B, C, D, E, F, R, and X, and a corresponding array with values 1, 2, 3, 4, 5, 6, and 7.
Linked Implementations

(a)

Val Size

(b)

R 2
A 3
B 1
C 0
D 0
E 0
F 0

Allocate child pointer space when node is created.
Sequential Implementations

List node values in the order they would be visited by a preorder traversal.

Saves space, but allows only sequential access.

Need to retain tree structure for reconstruction.

For binary trees: Use symbol to mark NULL links.

\[ AB/D//CEG///FH///I/// \]

Full binary trees: Mark leaf or internal.

\[ A'B'/DC'E'G/F'H'I \]

General trees: Mark end of each subtree.

\[ RAC)(D)E)))BF)))) \]
Convert to Binary Tree

Left Child/Right Sibling representation essentially stores a binary tree.

Use this process to convert any general tree to a binary tree.

A **forest** is a collection of one or more general trees.
Graphs

A **graph** $G = (V, E)$ consists of a set of **vertices** $V$, and a set of **edges** $E$, such that each edge in $E$ is a connection between a pair of vertices in $V$.

The number of vertices is written $|V|$, and the number of edges is written $|E|$.

A sequence of vertices $v_1, v_2, ..., v_n$ forms a **path** of length $n - 1$ if there exist edges from $v_i$ to $v_{i+1}$ for $1 \leq i < n$.

A path is **simple** if all vertices on the path are distinct.

A **cycle** is a path of length 3 or more that connects $v_i$ to itself.

A cycle is **simple** if the path is simple, except for the first and last vertices being the same.
An undirected graph is **connected** if there is at least one path from any vertex to any other. The maximal connected subgraphs of an undirected graph are called **connected components**.

A graph without cycles is **acyclic**.

A directed graph without cycles is a **directed acyclic graph** or DAG. A **free tree** is a connected, undirected graph with no simple cycles. Equivalently, a free tree is connected and has $|V - 1|$ edges.
Connected Components

Diagram showing connected components of a graph.
Graph Representations

**Adjacency Matrix:** $\Theta(|V|^2)$.

**Adjacency List:** $\Theta(|V| + |E|)$.
Implementation: Adjacency Matrix

class EdgeClass {
public:
    int v1, v2;
    EdgeClass(int in1, int in2) { v1 = in1; v2 = in2; }
};
typedef EdgeClass* Edge;

class Graph { // Adjacency matrix
private:
    int** matrix; // The edge matrix
    int numVertex; // Number of vertices
    int numEdge; // Number of edges
    bool* Mark; // The mark array
public:
    Graph(); // Constructor
    ~Graph(); // Destructor
    int n(); // Number of graph vertices
    int e(); // Number of edges for graph
    Edge first(int); // Get vertex first edge
    bool isEdge(Edge); // TRUE if this is an edge
    Edge next(Edge); // Get vertex next edge
    int v1(Edge); // Vertex edge comes from
    int v2(Edge); // Vertex edge goes to
    int weight(int, int); // Weight of edge
    int weight(Edge); // Weight of edge
    bool getMark(int); // Return a Mark value
    void setMark(int, bool); // Set a Mark value
};
Adjacency Matrix Functions

Edge Graph::first(int v) { // Get vertex first edge
    for (int i=0; i<numVertex; i++)
        if (matrix[v][i] != NOEDGE)
            return new EdgeClass(v, i);
    return NULL;
}

bool Graph::isEdge(Edge w) // TRUE if this is an edge
{ return (w != NULL) &&
    (matrix[w->v1][w->v2] != NOEDGE); }

Edge Graph::next(Edge w) { // Get vertex next edge
    for (int i=w->v2+1; i<numVertex; i++)
        if (matrix[w->v1][i] != NOEDGE)
            return new EdgeClass(w->v1, i);
    return NULL;
}

int Graph::weight(int i, int j) { // Return edge weight
    if (matrix[i][j] == NOEDGE) return INFINITY;
    else return matrix[i][j];
}

int Graph::weight(Edge w) { // Return weight of edge
    if ((w == NULL) || (matrix[w->v1][w->v2] == NOEDGE))
        return INFINITY;
    else return matrix[w->v1][w->v2];
}

int Graph::v1(Edge w) { return w->v1; } // Comes from
int Graph::v2(Edge w) { return w->v2; } // Goes to
bool Graph::getMark(int v) { return Mark[v]; }
void Graph::setMark(int v, bool val) { Mark[v] = val; }
Implementation: Adjacency List

class EdgeLink { // Singly-linked list node
public:
    int weight; // Edge weight
    int v1, v2; // Edge vertices
    EdgeLink* next; // Pointer to next list edge
    EdgeLink(int vt1, int vt2, int w, EdgeLink* nxt=NULL)
        { v1 = vt1; v2 = vt2; weight = w; next = nxt; }
    EdgeLink(EdgeLink* nxt =NULL) { next = nxt; }
};
typedef EdgeLink* Edge;

class Graph { // Graph class: Adjacency list
private:
    Edge* list; // The vertex list
    int numVertex, numEdge // Number of vertices, edges
    bool* Mark; // The mark array
public:
    Graph(); // Constructor
    ~Graph(); // Destructor
    int n(); // Number of vertices
    int e(); // Number of edges
    Edge first(int); // Get vertex first edge
    bool isEdge(Edge); // TRUE if this is an edge
    Edge next(Edge); // Get vertex next edge
    int v1(Edge); // Vertex edge comes from
    int v2(Edge); // Vertex edge goes to
    int weight(int, int); // Weight of edge
    int weight(Edge); // Weight of edge
    bool getMark(int); // Return a Mark value
    void setMark(int, bool); // Set a Mark value
};
Adjacency List Functions

Edge Graph::first(int v)  // Get first edge for vertex
{ return list[v]; }

bool Graph::isEdge(Edge w) // TRUE if this is an edge
{ return w != NULL; }

Edge Graph::next(Edge w) { // Get next edge for vertex
    if (w == NULL) return NULL;
    else return w->next;
}

int Graph::v1(Edge w)  // Return vertex it comes from
{ return w->v1; }

int Graph::v2(Edge w)  // Return vertex it goes to
{ return w->v2; }

int Graph::weight(int i, int j) { // Return edge weight
    for (Edge curr = list[i]; curr != NULL;
         curr = curr->next)
        if (curr->v2 == j) return curr->weight;
    return INFINITY;
}

int Graph::weight(Edge w) { // Return weight of edge
    if (w == NULL) return INFINITY;
    else return w->weight;
}
Graph Traversals

Some applications require visiting every vertex in the graph exactly once.

Application may require that vertices be visited in some special order based on graph topology.

Example: Artificial Intelligence

- Problem domain consists of many “states.”
- Need to get from Start State to Goal State.
- Start and Goal are typically not directly connected.

To insure visiting all vertices:

```cpp
void graph_traverse(Graph& G) {
    for (v=0; v<G.n(); v++)
        G.Mark[v] = UNVISITED; // Initialize mark bits
    for (v=0; v<G.n(); v++)
        if (G.Mark[v] == UNVISITED)
            do_traverse(G, v);
}
```
void DFS(Graph& G, int v) { // Depth first search
    PreVisit(G, v); // Take appropriate action
    G.setMark(v, VISITED);
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (G.getMark(G.v2(w)) == UNVISITED)
            DFS(G, G.v2(w));
    PostVisit(G, v); // Take appropriate action
}

Cost: \( \Theta(|V| + |E|) \).
Breadth First Search

Like DFS, but replace stack with a queue.

Visit the vertex’s neighbors before continuing deeper in the tree.

```cpp
void BFS(Graph& G, int start) {
    Queue Q(G.n());
    Q.enqueue(start);
    G.setMark(start, VISITED);
    while (!Q.isEmpty()) {
        int v = Q.dequeue();
        PreVisit(G, v); // Take appropriate action
        for (Edge w = G.first(v); G.isEdge(w); w=G.next(w))
            if (G.getMark(G.v2(w)) == UNVISITED) {
                G.setMark(G.v2(w), VISITED);
                Q.enqueue(G.v2(w));
            }
        PostVisit(G, v); // Take appropriate action
    }
}
```

(a) (b)
**Topological Sort**

Problem: Given a set of jobs, courses, etc. with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.

```cpp
void topsort(Graph& G) { // Topological sort: recursive
    for (int i=0; i<G.n(); i++) // Initialize Mark array
        G.setMark(i, UNVISITED);
    for (i=0; i<G.n(); i++) // Process all vertices
        if (G.getMark(i) == UNVISITED)
            tophelp(G, i); // Call helper function
}

void tophelp(Graph& G, int v) { // Helper function
    G.setMark(v, VISITED);
    // No PreVisit operation
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (G.getMark(G.v2(w)) == UNVISITED)
            tophelp(G, G.v2(w));
    printout(v); // PostVisit for Vertex v
}
```

#5BPrints in reverse order/#5D
void topsort(Graph& G) { // Topological sort: Queue
Queue Q(G.n());
int Count[G.n()];

for (int v=0; v<G.n(); v++) Count[v] = 0; // Init
for (v=0; v<G.n(); v++) // Process every edge
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
        Count[G.v2(w)]++; // Add to v2’s prereq count
for (v=0; v<G.n(); v++) // Initialize Queue
    if (Count[v] == 0) // Vertex has no prereqs
        Q.enqueue(v);
while (!Q.isEmpty()) { // Process the vertices
    int v = Q.dequeue(); // PreVisit for Vertex V
    printout(v);
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w)) {
        Count[G.v2(w)]--; // One less prerequisite
        if (Count[G.v2(w)] == 0) // Vertex is now free
            Q.enqueue(G.v2(w));
    }
}
}
Shortest Paths Problems

Input: A graph with weights or costs associated with each edge.

Output: The list of edges forming the shortest path.

Sample problems:
- Find the shortest path between two specified vertices.
- Find the shortest path from vertex $S$ to all other vertices.
- Find the shortest path between all pairs of vertices.

Our algorithms will actually calculate only distances.
Shortest Paths Definitions

d(A, B) is the **shortest distance** from vertex A to B.

w(A, B) is the **weight** of the edge connecting A to B.

- If there is no such edge, then w(A, B) = ∞.
Single Source Shortest Paths

Given start vertex $s$, find the shortest path from $s$ to all other vertices.

Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex $x$.

Problem: Shortest path to a vertex already processed might go through $x$.
Solution: Process vertices in order of distance from $s$. 
Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Process A</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>20</td>
<td>∞</td>
</tr>
<tr>
<td>Process C</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>Process B</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process D</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process E</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm: Array

void Dijkstra(Graph& G, int s) { // Use array
    int D[G.n()];
    for (int i=0; i<G.n(); i++) // Initialize
        D[i] = INFINITY;
    D[s] = 0;
    for (i=0; i<G.n(); i++) { // Process vertices
        int v = minVertex(G, D);
        if (D[v] == INFINITY) return; // Unreachable
        G.setMark(v, VISITED);
        for (Edge w = G.first(v); G.isEdge(w); w=G.next(w))
            if (D[G.v2(w)] > (D[v] + G.weight(w)))
                D[G.v2(w)] = D[v] + G.weight(w);
    }
}

int minVertex(Graph& G, int* D) { // Get mincost vertex
    int v; // Initialize v to any unvisited vertex;
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED) { v = i; break; }
    for (i++; i<G.n(); i++) // Now find smallest D value
        if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
            v = i;
    return v;
}

Approach 1: Scan the table on each pass for closest vertex.

Total cost: $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$. 

109
Dijkstra’s Algorithm: Priority Queue

class Elem { public: int vertex, dist; }
int key(Elem x) { return x.dist; }
void Dijkstra(Graph& G, int s) { // W/ priority queue
    int v; // The current vertex
    int D[G.n()]; // Distance array
    Elem temp;
    Elem E[G.e()]; // Heap array
    temp.dist = 0; temp.vertex = s;
    E[0] = temp; // Initialize heap
    heap H(E, 1, G.e()); // Create the heap
    for (int i=0; i<G.n(); i++) D[i] = INFINITY;
    D[s] = 0;
    for (i=0; i<G.n(); i++) { // Now, get distances
        do { temp = H.removemin(); v = temp.vertex; }
            while (G.getMark(v) == VISITED);
        G.setMark(v, VISITED);
        if (D[v] == INFINITY) return; // Rest unreachable
        for (Edge w = G.first(v); G.isEdge(w); w=G.next(w))
            if (D[G.v2(w)] > (D[v] + G.weight(w))) {
                D[G.v2(w)] = D[v] + G.weight(w); // Update D
                temp.dist = D[G.v2(w)]; temp.vertex = G.v2(w);
                H.insert(temp); // Insert new distance in heap
            }
    }
}

Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value. Must update priority queue for each edge.

Total cost: $\Theta((|V| + |E|)\log |V|)$. 

110
All Pairs Shortest Paths

For every vertex \( u, v \in V \), calculate \( d(u, v) \).

Could run Dijkstra’s Algorithm \( V \) times.

Better is Floyd’s Algorithm.

Define a **k-path** from \( u \) to \( v \) to be any path whose intermediate vertices all have indices less than \( k \).
Floyd’s Algorithm

void Floyd(Graph& G) {  // All-pairs shortest paths
    int D[G.n()][G.n()];  // Store distances
    for (int i=0; i<G.n(); i++) // Initialize D
        for (int j=0; j<G.n(); j++)
            D[i][j] = G.weight(i, j);
    for (int k=0; k<G.n(); k++) // Compute all k paths
        for (int i=0; i<G.n(); i++)
            for (int j=0; j<G.n(); j++)
                if (D[i][j] > (D[i][k] + D[k][j]))
                    D[i][j] = D[i][k] + D[k][j];
}
Minimum Cost Spanning Trees

Minimum Cost Spanning Tree (MST) Problem:
- **Input**: An undirected, connected graph $G$.
- **Output**: The subgraph of $G$ that 1) has minimum total cost as measured by summing the values for all of the edges in the subset, and 2) keeps the vertices connected.

![Graph with minimum spanning tree highlighted]
Prim’s MST Algorithm

```c
void Prim(Graph& G, int s) {  // Prim’s MST alg
    int D[G.n()];  // Distance vertex
    int V[G.n()];  // Who’s closest
    for (int i=0; i<G.n(); i++) {  // Initialize
        D[i] = INFINITY;
        D[s] = 0;
    }
    for (i=0; i<G.n(); i++) {  // Process vertices
        int v = minVertex(G, D);
        G.setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v);  // Add to MST
        if (D[v] == INFINITY) return;  // Rest unreachable
        for (Edge w = G.first(v); G.isEdge(w); w=G.next(w))
            if (D[G.v2(w)] > G.weight(w)) {
                D[G.v2(w)] = G.weight(w);  // Update distance,
                V[G.v2(w)] = v;          // who came from
            }
    }
}
```

```c
int minVertex(Graph& G, int* D) {  // Min cost vertex
    int v;  // Initialize v to any unvisited vertex;
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED) { v = i; break; }
    for (i=0; i<G.n(); i++)  // Now find smallest value
        if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
            v = i;
    return v;
}
```

This is an example of a **greedy** algorithm.

114
Alternative Prim’s Implementation

Like Dijkstra’s algorithm, we can implement Prim’s algorithm with a priority queue.

```cpp
void Prim(Graph& G, int s) { // W/ priority queue
    int v; // The current vertex
    int D[G.n()]; // Distance array
    int V[G.n()]; // Who’s closest
    Elem temp;
    Elem E[G.e()]; // Heap array
    temp.distance = 0; temp.vertex = s;
    E[0] = temp; // Initialize heap array
    heap H(E, 1, G.e()); // Create the heap
    for (int i=0; i<G.n(); i++) D[i] = INFINITY; // Init
    D[s] = 0;
    for (i=0; i<G.n(); i++) { // Now build MST
        do { temp = H.removemin(); v = temp.vertex; }
            while (G.getMark(v) == VISITED);
        G.setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v); // Add to MST
        if (D[v] == INFINITY) return; // Rest unreachable
        for (Edge w = G.first(v); G.isEdge(w); w=G.next(w))
            if (D[G.v2(w)] > G.weight(w)) { // Update D
                D[G.v2(w)] = G.weight(w);
                V[G.v2(w)] = v; // Who came from
                temp.distance = D[G.v2(w)];
                temp.vertex = G.v2(w);
                H.insert(temp); // Insert distance in heap
            }
    }
}
```
Proof of Prim’s MST Algorithm

Theorem 14.1 Prim’s algorithm produces a minimum cost spanning tree.

Proof by contradiction:
Order vertices by how they are added to the MST by Prim’s algorithm. $v_1, v_2, \ldots, v_n$.

Let edge $e_i$ connect $(v_x, v_{i+1})$, $x < i$.

Let $e_j$ be the lowest numbered (first) edge added by the algorithm such that the set of edges selected so far cannot be extended to form an MST for G.

Let $V_1 = (v_1, \ldots, v_j)$. Let $V_2 = (v_{j+1}, \ldots, v_n)$.
Kruskal’s MST Algorithm

Kruskel(Graph& G) { // Kruskal’s MST algorithm
    Gentree A(G.n()); // Equivalence class array
    Elem E[G.e()]; // Array of edges for min-heap
    int edgecnt = 0;
    for (int i=0; i<G.n(); i++) // Put edges on array
        for (Edge w = G.first(i);
            G.isEdge(w); w = G.next(w)) {
            E[edgecnt].weight = G.weight(w);
            E[edgecnt++].edge = w;
        }
    heap H(E, edgecnt, edgecnt); // Heapify the edges
    int numMST = G.n(); // Init w/ n equiv classes
    for (i=0; numMST>1; i++) { // Combine equiv classes
        Elem temp = H.removemin(); // Get next cheap edge
        Edge w = temp.edge;
        int v = G.v1(w); int u = G.v2(w);
        if (A.differ(v, u)) { // If different equiv classes
            A.UNION(v, u); // Combine equiv classes
            AddEdgetoMST(G.v1(w), G.v2(w)); // Add to MST
            numMST--; // One less MST
        }
    }
}

How do we compute function MSTof(v)?

Solution: Use Parent Pointer representation to merge equivalence classes.
Kruskal’s Algorithm Example

Time dominated by cost for initial edge sort.

Total cost: $\Theta(|V| + |E| \log |E|)$.

Initial

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

Step 1

Process edge (C, D)

Step 2

Process edge (E, F)

Step 3

Process edge (C, F)

118
Sorting

Each record contains a field called the **key**.
Linear order: comparison.

The Sorting Problem

Given a sequence of records \( R_1, R_2, \ldots, R_n \) with key values \( k_1, k_2, \ldots, k_n \), respectively, arrange the records into any order \( s \) such that records \( R_{s_1}, R_{s_2}, \ldots, R_{s_n} \) have keys obeying the property \( k_{s_1} \leq k_{s_2} \leq \ldots \leq k_{s_n} \).

Measures of cost:

- Comparisons
- Swaps
Insertion Sort

```c
void inssort(Elem* array, int n) { // Insertion Sort
    for (int i=1; i<n; i++) // Insert i’th record
        for (int j=i; (j>0) &&
            (key(array[j])<key(array[j-1])); j--)
            swap(array, j, j-1);
}
```

**Best Case:**

**Worst Case:**

**Average Case:**
Bubble Sort

void bubsrot(Elem* array, int n) { // Bubble Sort
for (int i=0; i<n-1; i++) // Bubble up i’th record
   for (int j=n-1; j>i; j--)
      if (key(array[j]) < key(array[j-1]))
         swap(array, j, j-1);
}

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>42</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>42</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>20</td>
<td>42</td>
<td>20</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>20</td>
<td>42</td>
<td>20</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>28</td>
<td>14</td>
<td>17</td>
<td>15</td>
<td>42</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>28</td>
<td>15</td>
<td>17</td>
<td>17</td>
<td>42</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>15</td>
<td>28</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>42</td>
</tr>
<tr>
<td>15</td>
<td>23</td>
<td>23</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
</tbody>
</table>

Best Case:

Worst Case:

Average Case:
Selection Sort

void selsort(Elem* array, int n) { // Selection Sort
    for (int i=0; i<n-1; i++) { // Select i’th record
        int lowindex = i; // Remember its index
        for (int j=n-1; j>i; j--) // Find the least value
            if (key(array[j]) < key(array[lowindex]))
                lowindex = j; // Put it in place
        swap(array, i, lowindex);
    }
}

Best Case:

Worst Case:

Average Case:
Pointer Swapping

(a) 

(b) 

Key = 42
Key = 5

Key = 42
Key = 5
## Exchange Sorting

### Summary

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Bubble</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparisons:</strong></td>
<td>Θ(n)</td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td><strong>Best Case</strong></td>
<td>Θ(n)</td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td><strong>Average Case</strong></td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td><strong>Worst Case</strong></td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td><strong>Swaps:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Best Case</strong></td>
<td>0</td>
<td>0</td>
<td>Θ(n)</td>
</tr>
<tr>
<td><strong>Average Case</strong></td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td><strong>Worst Case</strong></td>
<td>Θ(n^2)</td>
<td>Θ(n^2)</td>
<td>Θ(n)</td>
</tr>
</tbody>
</table>

All of these sorts rely on **exchanges** of adjacent records.

What is the average number of exchanges required?
Shellsort

void shellsort(Elem* array, int n) { // Shellsort
    for (int i=n/2; i>2; i/=2) // For each increment
        for (int j=0; j<i; j++) // Sort each sublist
            inssort2(&array[j], n-j, i);
    inssort2(array, n, 1);
}

// Version of Insertion Sort for varying increments
void inssort2(Elem* A, int n, int incr) {
    for (int i=incr; i<n; i+=incr)
        for (int j=i; (j>=incr) &&
             (key(A[j])<key(A[j-incr])); j-=incr)
            swap(A, j, j-incr);
}

59 20 17 13 28 14 23 83 36 98 11 70 65 41 42 15

36 20 11 13 28 14 23 15 59 98 17 70 65 41 42 83

28 14 11 13 36 20 17 15 59 41 23 70 65 98 42 83

11 13 17 14 23 15 28 20 36 41 42 70 59 83 65 98

11 13 14 15 17 20 23 28 36 41 42 59 65 70 83 98

$O(n^{1.5})$
Quicksort

Divide and Conquer: divide list into values less than pivot and values greater than pivot.

```c
void qsort(Elem* array, int i, int j) { // Quicksort
    int pivotindex = findpivot(array, i, j);
    swap(array, pivotindex, j); // Swap to end
    // k will be the first position in the right subarray
    int k = partition(array, i-1, j, key(array[j]));
    swap(array, k, j); // Put pivot in place
    if ((k-i) > 1) qsort(array, i, k-1); // Sort left
    if ((j-k) > 1) qsort(array, k+1, j); // Sort right
}

int findpivot(Elem* array, int i, int j)
{ return (i+j)/2; }
```
**Quicksort Partition**

```c
int partition(Elem* array, int l, int r, int pivot) {
    do { // Move the bounds inward until they meet
        while (key(array[++l]) < pivot); // Move right
        while (r && (key(array[--r]) > pivot)); // Move left
        swap(array, l, r); // Swap out-of-place vals
    } while (l < r); // Stop when they cross
    swap(array, l, r); // Reverse wasted swap
    return l; // Return first pos in right partition
}
```

<table>
<thead>
<tr>
<th>Initial</th>
<th>72 6 57 88 85 42 83 73 48 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 1</td>
<td>72 6 57 88 85 42 83 73 48 60</td>
</tr>
<tr>
<td>Swap 1</td>
<td>48 6 57 88 85 42 83 73 72 60</td>
</tr>
<tr>
<td>Pass 2</td>
<td>48 6 57 88 85 42 83 73 72 60</td>
</tr>
<tr>
<td>Swap 2</td>
<td>48 6 57 42 85 88 83 73 72 60</td>
</tr>
<tr>
<td>Pass 3</td>
<td>48 6 57 42 85 88 83 73 72 60</td>
</tr>
<tr>
<td>Swap 3</td>
<td>48 6 57 85 42 88 83 73 72 60</td>
</tr>
<tr>
<td>Reverse Swap</td>
<td>48 6 57 42 85 88 83 73 72 60</td>
</tr>
</tbody>
</table>

The cost for Partition is $\Theta(n)$. 127
Quicksort Example

<table>
<thead>
<tr>
<th>72</th>
<th>6</th>
<th>57</th>
<th>88</th>
<th>60</th>
<th>42</th>
<th>83</th>
<th>73</th>
<th>48</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>6</td>
<td>57</td>
<td>42</td>
<td>60</td>
<td>88</td>
<td>83</td>
<td>73</td>
<td>72</td>
<td>85</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>57</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>48</td>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pivot = 6

Pivot = 60

Pivot = 73

Pivot = 57

Pivot = 42

Pivot = 48

Pivot = 57

Pivot = 88

Pivot = 85

Pivot = 85

Final Sorted Array

<table>
<thead>
<tr>
<th>6</th>
<th>42</th>
<th>48</th>
<th>57</th>
<th>60</th>
<th>72</th>
<th>73</th>
<th>83</th>
<th>85</th>
<th>88</th>
</tr>
</thead>
</table>

128
Cost for Quicksort

Best Case: Always partition in half.

Worst Case: Bad partition.

Average Case:

\[
T(n) = n + 1 + \frac{1}{n-1} \sum_{k=1}^{n-1} (T(k) + T(n-k)) \\
= \Theta(n \log n)
\]

Optimizations for Quicksort:

- Better pivot.
- Use better algorithm for small sublists.
- Eliminate recursion.
Mergesort

List mergesort(List inlist) {
    if (inlist.length() <= 1) return inlist;;
    List l1 = half of the items from inlist;
    List l2 = other half of the items from inlist;
    return merge(mergesort(l1), mergesort(l2));
}
Mergesort Implementation

Mergesort is tricky to implement.

```c
void mergesort(Elem* array, Elem* temp,
               int left, int right) {
    int mid = (left+right)/2;
    if (left == right) return; // List of one ELEM
    mergesort(array, temp, left, mid); // Sort 1st half
    mergesort(array, temp, mid+1, right); // Sort 2nd half
    for (int i=left; i<=right; i++) // Copy to temp
        temp[i] = array[i];
    // Do the merge operation back to array
    int i1 = left; int i2 = mid + 1;
    for (int curr=left; curr<=right; curr++) {
        if (i1 == mid+1) // Left sublist exhausted
            array[curr] = temp[i2++];
        else if (i2 > right) // Right sublist exhausted
            array[curr] = temp[i1++];
        else if (key(temp[i1]) < key(temp[i2]))
            array[curr] = temp[i1++];
        else array[curr] = temp[i2++];
    }
}
```

Mergesort cost:

Mergesort is good for sorting linked lists.
void mergesort(Elem* array, Elem* temp, 
    int left, int right) {
    int i, j, k, mid = (left+right)/2;
    if (left == right) return;
    mergesort(array, temp, left, mid);  // Sort 1st half
    mergesort(array, temp, mid+1, right); // Sort 2nd half

    // Do merge operation. First, copy halves to temp.
    for (i=left; i<=mid; i++) temp[i] = array[i];
    for (j=1; j<=right-mid; j++)
        temp[right-j+1] = array[j+mid];
    // Merge sublists back to array
    for (i=left, j=right, k=left; k<=right; k++)
        if (key(temp[i]) < key(temp[j]))
            array[k] = temp[i++];
        else array[k] = temp[j--];
}
Heapsort

Heapsort uses a max-heap.

```c
void heapsort(Elem* array, int n) { // Heapsort
    heap H(array, n, n); // Build the heap
    for (int i=0; i<n; i++) // Now sort
        H.removemax(); // Value placed at end of heap
}
```

Cost of Heapsort:

Cost of finding $k$ largest elements:
Heapsort Example

Original Numbers

| 73 | 6 | 57 | 88 | 60 | 42 | 83 | 72 | 48 | 85 |

Build Heap

| 88 | 85 | 83 | 72 | 73 | 42 | 57 | 6 | 48 | 60 |

Remove 88

| 85 | 73 | 83 | 72 | 60 | 42 | 57 | 6 | 48 | 88 |

Remove 85

| 83 | 73 | 57 | 72 | 60 | 42 | 48 | 6 | 85 | 88 |

Remove 83

| 73 | 72 | 57 | 6 | 60 | 42 | 48 | 83 | 85 | 88 |
Binsort

A simple, efficient sort:

```c
for (i=0; i<n; i++)
    B[key(A[i])] = A[i];
```

Ways to generalize:

- Make each bin the head of a list.
- Allow more keys than records.

```c
void binsort(ELEM *A, int n) {
    list B[MaxKeyValue];
    for (i=0; i<n; i++) B[key(A[i])].append(A[i]);
    for (i=0; i<MaxKeyValue; i++)
        for (B[i].first(); B[i].isInList(); B[i].next())
            output(B[i].currValue());
}
```

Cost:
Radix Sort

Initial List: 27 91 1 97 17 23 84 28 72 5 67 25

First pass (on right digit)

Second pass (on left digit)

Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28
Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97
void radix(Elem* A, Elem* B, int n, int k, int r, int* count) {
    // Count[i] stores number of records in bin[i]

    for (int i=0, rtok=1; i<k; i++, rtok*=r) {// k digits
        for (int j=0; j<r; j++) count[j] = 0; // Init

        // Count number of records for each bin this pass
        for (j=0; j<n; j++) count[(key(A[j])/rtok)%r]++;

        // Index B: count[j] is index for last slot of j.
        for (j=1; j<r; j++) count[j] = count[j-1]+count[j];

        // Put recs into bins working from bottom of bin.
        // Since bins fill from bottom, j counts downwards
        for (j=n-1; j>=0; j--)
            B[--count[(key(A[j])/rtok)%r]] = A[j];

        for (j=0; j<n; j++) A[j] = B[j]; // Copy B to A
    }
}

Cost: $\Theta(nk + rk)$.

How do $n$, $k$ and $r$ relate?
Radix Sort Example

Initial Input: Array A

<table>
<thead>
<tr>
<th></th>
<th>27</th>
<th>91</th>
<th>1</th>
<th>97</th>
<th>17</th>
<th>23</th>
<th>84</th>
<th>28</th>
<th>72</th>
<th>5</th>
<th>67</th>
<th>25</th>
</tr>
</thead>
</table>

First pass values for Count.
rtok = 1.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

Count array:
Index positions for Array B.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>11</th>
<th>12</th>
<th>12</th>
</tr>
</thead>
</table>

End of Pass 1: Array A.

<table>
<thead>
<tr>
<th></th>
<th>91</th>
<th>1</th>
<th>72</th>
<th>23</th>
<th>84</th>
<th>5</th>
<th>25</th>
<th>27</th>
<th>97</th>
<th>17</th>
<th>67</th>
<th>28</th>
</tr>
</thead>
</table>

Second pass values for Count.
rtok = 10.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

Count array:
Index positions for Array B.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>7</th>
<th>7</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
</table>

End of Pass 2: Array A.

|   | 1 | 5 | 17 | 23 | 25 | 27 | 28 | 67 | 72 | 84 | 91 | 97 |
# Empirical Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10K</th>
<th>30K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert. Sort</td>
<td>.00200</td>
<td>.1833</td>
<td>18.13</td>
<td>1847.0</td>
<td>16544.0</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>.00233</td>
<td>.2267</td>
<td>22.47</td>
<td>2274.0</td>
<td>20452.0</td>
</tr>
<tr>
<td>Selec. Sort</td>
<td>.00167</td>
<td>.0967</td>
<td>2.17</td>
<td>900.3</td>
<td>8142.0</td>
</tr>
<tr>
<td>Shellsort</td>
<td>.00233</td>
<td>.0600</td>
<td>1.00</td>
<td>17.0</td>
<td>59.0</td>
</tr>
<tr>
<td>Shellsort/O</td>
<td>.00233</td>
<td>.0500</td>
<td>.93</td>
<td>16.3</td>
<td>65.0</td>
</tr>
<tr>
<td>QSort</td>
<td>.00367</td>
<td>.0500</td>
<td>.63</td>
<td>7.3</td>
<td>24.0</td>
</tr>
<tr>
<td>QSort/O</td>
<td>.00200</td>
<td>.0300</td>
<td>.43</td>
<td>5.7</td>
<td>18.0</td>
</tr>
<tr>
<td>Merge</td>
<td>.00700</td>
<td>.0700</td>
<td>.87</td>
<td>10.7</td>
<td>35.0</td>
</tr>
<tr>
<td>Merge/O</td>
<td>.00133</td>
<td>.0267</td>
<td>.37</td>
<td>5.0</td>
<td>16.0</td>
</tr>
<tr>
<td>Heapsort</td>
<td>.00900</td>
<td>.1767</td>
<td>2.67</td>
<td>36.3</td>
<td>122.0</td>
</tr>
<tr>
<td>Rad Sort/1</td>
<td>.02433</td>
<td>.2333</td>
<td>2.30</td>
<td>23.3</td>
<td>69.0</td>
</tr>
<tr>
<td>Rad Sort/4</td>
<td>.00700</td>
<td>.0600</td>
<td>.60</td>
<td>6.0</td>
<td>17.0</td>
</tr>
<tr>
<td>Rad Sort/8</td>
<td>.00967</td>
<td>.0333</td>
<td>.30</td>
<td>3.0</td>
<td>8.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10K</th>
<th>100K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert. Sort</td>
<td>.0002</td>
<td>.0170</td>
<td>1.68</td>
<td>168.8</td>
<td>23382.0</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>.0003</td>
<td>.0257</td>
<td>2.55</td>
<td>257.2</td>
<td>41874.0</td>
</tr>
<tr>
<td>Selec. Sort</td>
<td>.0003</td>
<td>.0273</td>
<td>2.65</td>
<td>267.5</td>
<td>40393.0</td>
</tr>
<tr>
<td>Shellsort</td>
<td>.0003</td>
<td>.0027</td>
<td>0.12</td>
<td>1.9</td>
<td>40.0</td>
</tr>
<tr>
<td>Shellsort/O</td>
<td>.0003</td>
<td>.0060</td>
<td>0.11</td>
<td>1.8</td>
<td>33.0</td>
</tr>
<tr>
<td>QSort</td>
<td>.0004</td>
<td>.0057</td>
<td>0.08</td>
<td>0.9</td>
<td>12.0</td>
</tr>
<tr>
<td>QSort/O</td>
<td>.0002</td>
<td>.0040</td>
<td>0.06</td>
<td>0.8</td>
<td>10.0</td>
</tr>
<tr>
<td>Merge</td>
<td>.0009</td>
<td>.0130</td>
<td>0.17</td>
<td>2.3</td>
<td>30.0</td>
</tr>
<tr>
<td>Merge/O</td>
<td>.0003</td>
<td>.0067</td>
<td>0.11</td>
<td>1.5</td>
<td>21.0</td>
</tr>
<tr>
<td>Heapsort</td>
<td>.0010</td>
<td>.0173</td>
<td>0.26</td>
<td>3.5</td>
<td>49.0</td>
</tr>
<tr>
<td>Rad Sort/1</td>
<td>.0123</td>
<td>.1197</td>
<td>1.21</td>
<td>12.5</td>
<td>135.0</td>
</tr>
<tr>
<td>Rad Sort/4</td>
<td>.0035</td>
<td>.0305</td>
<td>0.30</td>
<td>3.2</td>
<td>34.0</td>
</tr>
<tr>
<td>Rad Sort/8</td>
<td>.0047</td>
<td>.0183</td>
<td>0.16</td>
<td>1.6</td>
<td>18.0</td>
</tr>
</tbody>
</table>
Sorting Lower Bound

Want to prove a lower bound for all possible sorting algorithms.

Sorting is $O(n \log n)$.

Sorting I/O takes $\Omega(n)$ time.

Will now prove $\Omega(n \log n)$ lower bound.

Form of proof:
- Comparison based sorting can be modeled by a binary tree.
- The tree must have $\Omega(n!)$ leaves.
- The tree must be $\Omega(n \log n)$ levels deep.
### Decision Trees

There are $n!$ permutations, and at least 1 node for each permutation.

A tree with $n$ nodes has at least $\log n$ levels.

Where is the worst case in the decision tree?

$\log n! = \Omega(n \log n)$. 

---

141
Primary vs. Secondary Storage

**Primary Storage:** Main memory (RAM)

**Secondary Storage:** Peripheral devices
- Disk Drives
- Tape Drives

<table>
<thead>
<tr>
<th>Medium</th>
<th>Price</th>
<th>Price per Mbyte</th>
</tr>
</thead>
<tbody>
<tr>
<td>32MB RAM</td>
<td>$225</td>
<td>$7.00/MB</td>
</tr>
<tr>
<td>1.4MB floppy disk</td>
<td>$.50</td>
<td>$0.36/MB</td>
</tr>
<tr>
<td>2.1GB disk drive</td>
<td>$210</td>
<td>$0.10/MB</td>
</tr>
<tr>
<td>1GB JAZ cassette</td>
<td>$100</td>
<td>$0.10/MB</td>
</tr>
<tr>
<td>2GB cartridge tape</td>
<td>$20</td>
<td>$0.01/MB</td>
</tr>
</tbody>
</table>

RAM is usually **volatile**.

RAM is about 1/4 million times faster than disk.
Golden Rule of File Processing

Minimize the number of disk accesses!

1. Arrange information so that you get what you want with few disk accesses.
2. Arrange information so minimize future disk accesses.

An organization for data on disk is often called a **file structure**.

Disk based space/time tradeoff: Compress information to save processing time by reducing disk accesses.
Disk Drives

(a) Boom (arm)
- Platters
- Spindle
- Read/Write Heads

(b) Track

CD-ROM: Spiral with equally spaced dots, variable speed rotation.

Spacing loses 1/2 of potential data density.

Bits of data Intersector Gaps Sectors

144
A **sector** is the basic unit of I/O.

**Interleaving factor:** Physical distance between logically adjacent sectors on a track, to allow for processing of sector data.

**Locality of Reference:** If a record is read from disk, the next request is likely to come from near the same place in the file.

**Cluster:** Smallest unit of file allocation – several sectors.

**Extent:** A group of physically contiguous clusters.

**Internal Fragmentation:** Wasted space within a sector if record size does not match sector size, or wasted space within a cluster if file size is not a multiple of cluster size.
Factors affecting disk access time

1. **Seek time**: time for I/O head to reach desired track. Largely determined by distance between I/O head and desired track.

   Seek time: \( f(n) = t \times n + s \) where \( t \) is time to traverse one track and \( s \) is startup time for the I/O head.

2. **Rotational delay or latency**: time for data to rotate to I/O head position.

   At 3600 RPM, one half rotation of the disk: \( 16.7/2 \) msec = 8.3 msec.

3. **Transfer time**: time for data to move under the I/O head.

   Number of sectors read / Number of sectors per track * 16.7 msec
Disk Access Cost Example

675 Mbyte disk drive
- 15 platters ⇒ 45 Mbyte/platter
- 612 tracks/platter
- 150 sectors/track ⇒ 512 bytes/sector
- 8 sectors/cluster (4K bytes/cluster) ⇒ 18 clusters/track
- Interleaving factor of 3 ⇒ 3 revolutions to read one track (50.1 msec)

How long to read a file of 128 Kbytes divided into 256 records of 512 bytes?

Number of Clusters:

If file fills minimum number of tracks:
- 150 sectors of one track, 106 of the next

Total time:
\[
\frac{612}{3} \times 0.08 + 3 + 3.5 \times 16.7 + 0.08 + 3 +
3.5 \times 16.7 = 139.3 \text{ msec.}
\]

If clusters are spread randomly across disk:
\[
32 \times \left( \frac{\frac{612}{3}}{3} \times 0.08 + 3 + \frac{16.7}{2} + \frac{24}{150} \times 16.7 \right)
= 32 \times 30.3 = 969.6 \text{ msec.}
\]
Magnetic Tape

Example: 9 track tape at 6250 bytes per inch (bpi).
At 2400 feet, this yields 170 Mbytes for $20, or $0.12/Mbyte.

Workstation/PC cartridge tape is similar.

Magnetic tape requires **sequential** access.

Magnetic tape has two speeds:
- High speed for “skipping.”
- Low speed for “reading.”

**Interblock Gap** is space required for I/O head to recognize beginning of a record at high speed. This is a significant amount of space compared to typical record size.

**Blocking Factor**: Number of records in a block between gaps.
Buffers

Read time for one track:

\[ \frac{612}{3} \times 0.08 + 3 + 3.5 \times 16.7 = 77.8 \text{ msec.} \]

Read time for one sector:

\[ \frac{612}{3} \times 0.08 + 3 + 16.7/2 + 16.7/150 = 27.8 \text{ msec.} \]

Read time for one byte:

\[ \frac{612}{3} \times 0.08 + 3 + 16.7/2 = 27.7 \text{ msec.} \]

Nearly all disk drives read/write one sector at every I/O access.
- Also called a page.

The information in a sector is stored in a buffer or cache.

If next I/O access is to same buffer, then no need to go to disk.

There are usually one or more input buffers and one or more output buffers.
Buffer Pools

A series of buffers used by an application to cache disk data is called a buffer pool.

Virtual memory uses a buffer pool to imitate greater RAM memory by actually storing information on disk and “swapping” between disk and RAM.

Organization for buffer pools: which one to use next?

• First-in, First-out: Use the first one on the queue.
• Least Frequently Used (LFU): Count buffer accesses, pick the least used.
• Least Recently Used (LRU): Keep buffers on linked list. When a buffer is accessed, bring to front. Reuse the one at the end.

Double Buffering: Read data from disk for next buffer while CPU is processing previous buffer.
Programmer’s View of Files

Logical view of files:
- An array of bytes.
- A file pointer marks the current position.

Three fundamental operations:
- Read bytes from current position (move file pointer).
- Write bytes to current position (move file pointer).
- Set file pointer to specified byte position.
C/C++ File Functions

FILE *fopen(char *filename, char *mode);
void fclose(FILE *stream);

Mode examples:
- "rb": open a binary file, read-only.
- "w+t": create a text file for reading and writing.

size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
if(numrec !=
    fread(recarr, sizeof rec, numrec, myfile))
    its_an_error();

size_t fwrite(void *ptr, size_t size, size_t n, FILE *stream);

int fseek(FILE *stream, long offset, int whence);
where whence is one of:
- SEEK_SET: Offset from file beginning.
- SEEK_CUR: Offset from current position.
- SEEK_END: Offset from end of file.
External Sorting

Problem: Sorting data sets too large to fit in main memory.

- Assume data stored on disk drive.

To sort, portions of the data must be brought into main memory, processed, and returned to disk.

An external sort should minimize disk accesses.
Model of External Computation

Secondary memory is divided into equal-sized blocks (512, 2048, 4096 or 8192 bytes are typical sizes).

The basic I/O operation transfers the contents of one disk block to/from main memory.

Under certain circumstances, reading blocks of a file in sequential order is more efficient. (When?)

Typically, the time to perform a single block I/O operation is sufficient to Quicksort the contents of the block.

Thus, our primary goal is to minimize the number of block I/O operations.

Most workstations today must do all sorting on a single disk drive.
Key Sorting

Often records are large while keys are small.

- Ex: Payroll entries keyed on ID number.

Approach 1: Read in entire records, sort them, then write them out again.

Approach 2: Read only the key values, store with each key the location on disk of its associated record.

If necessary, after the keys are sorted the records can be read and re-written in sorted order.
External Sort: Simple Mergesort

Quicksort requires random access to the entire set of records.

Better: Modified Mergesort algorithm

- Process \( n \) elements in \( \Theta(\log n) \) passes.

1. Split the file into two files.
2. Read in a block from each file.
3. Take first record from each block, output them in sorted order.
4. Take next record from each block, output them to a second file in sorted order.
5. Repeat until finished, alternating between output files. Read new input blocks as needed.
6. Repeat steps 2-5, except this time the input files have groups of two sorted records that are merged together.
7. Each pass through the files provides larger and larger groups of sorted records.

A group of sorted records is called a **run**.
Problems with Simple Mergesort

Is each pass through input and output files sequential?

What happens if all work is done on a single disk drive?

How can we reduce the number of Mergesort passes?

In general, external sorting consists of two phases:

1. Break the file into initial runs.
2. Merge the runs together into a single sorted run.
Breaking a file into runs

General approach:

- Read as much of the file into memory as possible.
- Perform and in-memory sort.
- Output this group of records as a single run.
Replacement Selection

1. Break available memory into an array for the heap, an input buffer and an output buffer.
2. Fill the array from disk.
3. Make a min-heap.
4. Send the smallest value (root) to the output buffer.
5. If the next key in the file is greater than the last value output, then
   Replace the root with this key.
   else
   Replace the root with the last key in the array.
   Add the next record in the file to a new heap (actually, stick it at the end of the array).
## Example of Replacement Selection

<table>
<thead>
<tr>
<th>Input</th>
<th>Memory</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>![Tree with numbers 12, 19, 31, 25, 21, 56, 40]</td>
<td>12</td>
</tr>
<tr>
<td>29</td>
<td>![Tree with numbers 16, 19, 31, 25, 21, 56, 40]</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>![Tree with numbers 19, 21, 31, 25, 29, 56, 40]</td>
<td>19</td>
</tr>
<tr>
<td>35</td>
<td>![Tree with numbers 21, 25, 31, 40, 29, 56, 14]</td>
<td>21</td>
</tr>
</tbody>
</table>

160
Use double buffering to overlap input, processing and output.

How many disk drives for greatest advantage?

Snowplow argument:

- A snowplow moves around a circular track onto which snow falls at a steady rate.
- At any instant, there is a certain amount of snow $S$ on the track. Some falling snow comes in front of the plow, some behind.
- During the next revolution of the snowplow, all of this is removed, plus $1/2$ of what falls during that revolution.
- Thus, the plow removes $2S$ amount of snow.

Is this always true?
Simple Mergesort may not be Best

Simple Mergesort: Place the runs into two files.
  • Merge the first two runs to output file, then next two runs, etc.

This process is repeated until only one run remains.
  • How many passes for $r$ initial runs?

Is there benefit from sequential reading?

Is working memory well used?

Need a way to reduce the number of passes.
**Multiway Merge**

With replacement selection, each initial run is several blocks long.

Assume that each run is placed in a separate disk file.

We could then read the first block from each file into memory and perform an $r$-way merge.

When a buffer becomes empty, read a block from the appropriate run file.

Each record is read only once from disk during the merge process.

In practice, use only one file and seek to appropriate block.

```
Input Runs

| 5 10 15 |...
|--------|
| 6 7 23 |...
|        |
| 12 18 20 |...

Output Buffer

| 5 6 7 10 12 |...
```
Limits to Single Pass Multiway Merge

Assume working memory is $b$ blocks in size.

How many runs can be processed at one time?
The runs are $2b$ blocks long (on average).

How big a file can be merged in one pass?

Larger files will need more passes – but the run size grows quickly!

This approach trades $\Theta(\log b)$ (possibly) sequential passes for a single or a very few random (block) access passes.
General Principals of External Sorting

In summary, a good external sorting algorithm will seek to do the following:

- Make the initial runs as long as possible.
- At all stages, overlap input, processing and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing.
- If possible, use additional disk drives for more overlapping of processing with I/O, and allow for more sequential file processing.
Search

Given: Distinct keys $k_1$, $k_2$, ... $k_n$ and collection $T$ of $n$ records of the form

$$(k_1, I_1), (k_2, I_2), ..., (k_n, I_n)$$

where $I_j$ is information associated with key $k_j$ for $1 \leq j \leq n$.

Search Problem: For key value $K$, locate the record $(k_j, I_j)$ in $T$ such that $k_j = K$.

Searching is a systematic method for locating the record (or records) with key value $k_j = K$.

A successful search is one in which a record with key $k_j = K$ is found.

An unsuccessful search is one in which no record with $k_j = K$ is found (and presumably no such record exists).
Approaches to Search

1. Sequential and list methods (lists, tables, arrays).
2. Direct access by key value (hashing).
3. Tree indexing methods.
Searching Ordered Arrays

Sequential Search

Binary Search

```c
int binary(int K, int* array, int left, int right) {
    // Return pos of ELEM in array (if any) with value K
    int l = left-1;
    int r = right+1; // l, r beyond bounds of array
    while (l+1 != r) { // Stop when l and r meet
        int i = (l+r)/2; // Look at middle of subarray
        if (K < array[i]) r = i; // In left half
        if (K == array[i]) return i; // Found it
        if (K > array[i]) l = i; // In right half
    }
    return UNSUCCESSFUL; // Search value not in array
}
```

```
<table>
<thead>
<tr>
<th>Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>11</td>
<td>13</td>
<td>21</td>
<td>26</td>
<td>29</td>
<td>36</td>
<td>40</td>
<td>41</td>
<td>45</td>
<td>51</td>
<td>54</td>
<td>56</td>
<td>65</td>
<td>72</td>
<td>77</td>
<td>83</td>
</tr>
</tbody>
</table>
```

Dictionary Search
Lists Ordered by Frequency

Order lists by (expected) frequency of occurrence.

- Perform sequential search.

Cost to access first record: 1
Cost to access second record: 2

Expected search cost:

\[ \overline{C}_n = 1p_1 + 2p_2 + \ldots + np_n \]

Example: all records have equal frequency

\[ \overline{C}_n = \sum_{i=1}^{n} \frac{i}{n} = \frac{n+1}{2}. \]

Example: Exponential frequency

\[ p_i = \begin{cases} 
1/2^i & \text{if } 1 \leq i \leq n - 1 \\
1/2^{n-1} & \text{if } i = n 
\end{cases} \]

\[ \overline{C}_n \approx \sum_{i=1}^{n} (i/2^i) \approx 2. \]
Zipf Distributions

Applications:

- Distribution for frequency of word usage in natural languages.
- Distribution for populations of cities, etc.

Definition: Zipf frequency for item $i$ in the distribution for $n$ records as $1/i\mathcal{H}_n$.

$$\overline{C}_n = \sum_{i=1}^{n} \frac{i}{i\mathcal{H}_n} = \frac{n}{\mathcal{H}_n} \approx \frac{n}{\log_e n}$$

80/20 rule: 80% of the accesses are to 20% of the records.

For distributions following the 80/20 rule,

$$\overline{C}_n \approx 0.122n.$$
Self-Organizing Lists

Self-organizing lists modify the order of records within the list based on the actual pattern of record access.

Self-organizing lists use a rule called a **heuristic** for deciding how to reorder the list. These heuristics are similar to the rules for managing buffer pools.

- **Order by actual historical frequency of access.** (Similar to LFU buffer pool replacement strategy.)
- When a record is found, swap it with the first record on list.
- **Move-to-Front**: When a record is found, move it to the front of the list.
- **Transpose**: When a record is found, swap it with the record ahead of it.
Example of Self-Organizing Tables

Application: Text compression.

Keep a table of words already seen, organized via Move-to-Front Heuristic.

If a word not yet seen, send the word.
Otherwise, send the (current) index in the table.

The car on the left hit the car I left.

The car on 3 left hit 3 5 I 5.

This is similar in spirit to Ziv-Lempel coding.
Searching in Sets

For dense sets (small range, many elements in set):

Can use logical bit operators.

Example: To find all primes that are odd numbers, compute

0011010100010100 & 0101010101010101

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 0 0 1 1 0 1 0 1 0 0 0 1 0 1 0 0
```
**Hashing**

**Hashing**: The process of mapping a key value to a position in a table.

A **hash function** maps key values to positions. It is denoted by $h$.

A **hash table** is an array that holds the records. It is denoted by $T$.

The hash table has $M$ slots, indexed from 0 to $M - 1$.

For any value $K$ in the key range and some hash function $h$,

$$h(K) = i, \ 0 \leq i < M,$$

such that $\text{key}(T[i]) = K$. 
Hashing (continued)

Hashing is appropriate only for sets (no duplicates).

Good for both in-memory and disk based applications.

Answers the question “What record, if any, has key value $K$?”

Example: Store the $n$ records with keys in range 0 to $n - 1$.

- Store the record with key $i$ in slot $i$.
- Use hash function $h(K) = K$. 
Collisions

More reasonable example:

- Store about 1000 records with keys in range 0 to 16,383.
- Impractical to keep a hash table with 16,384 slots.
- We must devise a hash function to map the key range to a smaller table.

Given: hash function $h$ and keys $k_1$ and $k_2$. $\beta$ is a slot in the hash table.
If $h(k_1) = \beta = h(k_2)$, then $k_1$ and $k_2$ have a collision at $\beta$ under $h$.

Search for the record with key $K$:
1. Compute the table location $h(K)$.
2. Starting with slot $h(K)$, locate the record containing key $K$ using (if necessary) a collision resolution policy.

Collisions are inevitable in most applications.
- Example: 23 people are likely to share a birthday.
Hash Functions

A hash function MUST return a value within the hash table range.

To be practical, a hash function SHOULD evenly distribute the records stored among the hash table slots.

Ideally, the hash function should distribute records with equal probability to all hash table slots. In practice, success depends on the distribution of the actual records stored.

If we know nothing about the incoming key distribution, evenly distribute the key range over the hash table slots while avoiding obvious opportunities for clustering.

If we have knowledge of the incoming distribution, use a distribution-dependant hash function.
Example Hash Functions

```c
int h(int x) {
    return(x % 16);
}
```

This function is entirely dependant on the lower 4 bits of the key.

**Mid-square method**: square the key value, take the middle \( r \) bits from the result for a hash table of \( 2^r \) slots.

Sum the ASCII values of the letters and take results modulo \( M \).

```c
int h(char x[10]) {
    int i, sum;
    for (sum=0, i=0; i<10; i++)
        sum += (int) x[i];
    return(sum % M);
}
```
ELF Hash


```c
int ELFhash(char* key) {
    unsigned long h = 0;
    while(*key) {
        h = (h << 4) + *key++;
        unsigned long g = h & 0xF0000000L;
        if (g) h ^= g >> 24;
        h &= ~g;
    }
    return h % M;
}
```
Open Hashing

What to do when collisions occur? **Open hashing** treats each hash table slot as a bin.

![Diagram of a hash table with collisions and resolutions](image)
Bucket Hashing

Divide the hash table slots into buckets.
- Example: 8 slots/bucket.

Include an overflow bucket.

Records hash to the first slot of the bucket, and fill bucket. Go to overflow if necessary.

When searching, first check the proper bucket. Then check the overflow.
Closed Hashing

Closed hashing stores all records directly in the hash table.

Each record $i$ has a **home position** $h(k_i)$.

If $i$ is to be inserted and another record already occupies $i$’s home position, then another slot must be found to store $i$.

The new slot is found by a **collision resolution policy**.

Search must follow the same policy to find records not in their home slots.
Collision Resolution

During insertion, the goal of collision resolution is to find a free slot in the table.

**Probe Sequence**: the series of slots visited during insert/search by following a collision resolution policy.

Let $\beta_0 = h(K)$. Let $(\beta_0, \beta_1, \ldots)$ be the series of slots making up the probe sequence.

```c
void hashInsert(Elem R) { // Insert R into hash table T
    int home; // Home position for R
    int pos = home = h(key(R)); // Initial pos on sequence
    for (int i=1; key(T[pos]) != EMPTY; i++) {
        pos = (home + p(key(R), i)) % M; // Next slot
        if (key(T[pos]) == key(R)) ERROR; // No duplicates
    }
    T[pos] = R; // Insert R
}
```

```c
Elem hashSearch(int K) { // Search for record w/ key K
    int home; // Home position for K
    int pos = home = h(K); // Initial pos on sequence
    for (int i = 1; (key(T[pos]) != K) &&
        (key(T[pos]) != EMPTY); i++)
        pos = (home + p(K, i)) % M; // Next pos on sequence
    if (key(T[pos]) == K) return T[pos]; // Found it
    else return UNSUCCESSFUL; // K not in hash table
}
```
Linear Probing

Use the probe function

```c
int p(int K, int i) { return i; }
```

This is called **linear probing**.

Linear probing simply goes to the next slot in the table.

If the bottom is reached, wrap around to the top.

To avoid an infinite loop, one slot in the table must always be empty.
Primary Clustering: Records tend to cluster in the table under linear probing since the probabilities for which slot to use next are not the same.
**Improved Linear Probing**

Instead of going to the next slot, skip by some constant $c$.

**Warning:** Pick $M$ and $c$ carefully.

The probe sequence SHOULD cycle through all slots of the table.

Pick $c$ to be relatively prime to $M$.

There is still some clustering.

- Example: $c = 2$. $h(k_1) = 3$. $h(k_2) = 5$.
- The probe sequences for $k_1$ and $k_2$ are linked together.
Pseudo Random Probing

The ideal probe function would select the next slot on the probe sequence at random.

An actual probe function cannot operate randomly. (Why?)

**Pseudo random probing:**

- Select a (random) permutation of the numbers from 1 to $M - 1$:
  
  $$r_1, r_2, \ldots, r_{M-1}$$

- All insertions and searches use the same permutation.

Example: Hash table of size $M = 101$

- $r_1 = 2, r_2 = 5, r_3 = 32$.
- $h(k_1) = 30, h(k_2) = 28$.

- Probe sequence for $k_1$ is:
- Probe sequence for $k_2$ is:
Quadratic Probing

Set the $i$’th value in the probe sequence as

$$(h(K) + i^2) \mod M.$$ 

Example: $M = 101$.
- $h(k_1) = 30$, $h(k_2) = 29$.
- Probe sequence for $k_1$ is:
- Probe sequence for $k_2$ is:
Double Hashing

Pseudo random probing eliminates primary clustering.

If two keys hash to same slot, they follow the same probe sequence. This is called **secondary clustering**.

To avoid secondary clustering, need a probe sequence to be a function of the original key value, not just the home position.

**Double hashing:**

\[ p(K, i) = i \times h_2(K) \text{ for } 0 \leq i \leq M - 1. \]

Be sure that all probe sequence constants are relatively prime to \( M \).

Example: Hash table of size \( M = 101 \)

- \( h(k_1) = 30, h(k_2) = 28, h(k_3) = 30 \).
- \( h_2(k_1) = 2, h_2(k_2) = 5, h_2(k_3) = 5 \).
- Probe sequence for \( k_1 \) is:
- Probe sequence for \( k_2 \) is:
- Probe sequence for \( k_3 \) is:
Analysis of Closed Hashing

The **load factor** is $\alpha = \frac{N}{M}$ where $N$ is the number of records currently in the table.
Deletion

1. Deleting a record must not hinder later searches.
2. We do not want to make positions in the hash table unusable because of deletion.

Both of these problems can be resolved by placing a special mark in place of the deleted record, called a **tombstone**.

A tombstone will not stop a search, but that slot can be used for future insertions.

Unfortunately, tombstones do add to the average path length.

Solutions:

1. Local reorganizations to try to shorten the average path length.
2. Periodically rehash the table (by order of most frequently accessed record).
Indexing

Goals:

• Store large files.
• Support multiple search keys.
• Support efficient insert, delete and range queries.

Entry sequenced file: Order records by time of insertion.

Use sequential search.

Index file: Organized, stores pointers to actual records.

Primary key: A unique identifier for records. May be inconvenient for search.

Secondary key: an alternate search key, often not unique for each record. Often used for search key.
Linear Indexing

**Linear Index**: an index file organized as a simple sequence of key/record pointer pairs where the key values are in sorted order.

If the index is too large to fit in main memory, a second level index may be used.

Linear indexing is good for searching variable length records.

Linear indexing is poor for insert/delete.

---

**Linear Index**

```
| 37 | 42 | 52 | 73 | 98 |
```

| 73 | 52 | 98 | 37 | 42 |

**Database Records**

```
1 2003 5894 10528
```

**Second Level Index**

```
1 2001 2003 5688 5894 9942 10528 10984
```

**Linear Index: Disk Pages**
Inverted list is another term for a secondary index. A secondary key is associated with a primary key, which in turn locates the record.

<table>
<thead>
<tr>
<th>Secondary Key</th>
<th>Primary Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>AA10</td>
</tr>
<tr>
<td></td>
<td>AB12</td>
</tr>
<tr>
<td></td>
<td>AB39</td>
</tr>
<tr>
<td></td>
<td>FF37</td>
</tr>
<tr>
<td>Smith</td>
<td>AX33</td>
</tr>
<tr>
<td></td>
<td>AX35</td>
</tr>
<tr>
<td></td>
<td>ZX45</td>
</tr>
<tr>
<td>Zukowski</td>
<td>ZQ99</td>
</tr>
</tbody>
</table>
Inverted List (Continued)

Secondary Key | Index
---|---
Jones | 0
Smith | 1
Zukowski | 3

Primary Key | Next
---|---
AA10 | 4
AX33 | 6
ZX45 | 2
ZQ99 | 2
AB12 | 5
AB39 | 7
AX35 | 2
FF37 | 2
Tree Indexing

Linear index is poor for insertion/deletion.

Tree index can efficiently support all desired operations:

- Insert/delete
- Multiple search keys
- Key range search

Storing a tree index on disk causes additional problems:

1. Tree must be balanced.
2. Each path from root to a leaf should cover few disk pages.
A 2-3 Tree has the following properties:

1. A node contains one or two keys.
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys).
3. All leaves are at the same level in the tree, so the tree is always height balanced.

The 2-3 Tree also has a search tree property analogous to the BST.

The advantage of the 2-3 Tree over the BST is that it can be updated at low cost.

---

Diagram of a 2-3 Tree:

```
            18  33
           /    |
         12     |
        /     |
       10  15  |
           /   |
          20  21  |
          /     |
         24     31
          /     |
         45  47  50  52
```
2-3 Tree Insertion

Always insert at leaf node.
2-3 Tree Splitting

(a) 20

(b) 23

(c) 23

19 21 24 31

18

12 15 19 21

20

33

30

48

24 31 45 47 50 52

\[\text{All operations are local to original search path.}\]
B-Trees

The B-Tree is an extension of the 2-3 Tree.

The B-Tree is now the standard file organization for applications requiring insertion, deletion and key range searches.

1. B-Trees are always balanced.
2. B-Trees keep related records on a disk page, which takes advantage of locality of reference.
3. B-Trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.
B-Trees (Continued)

A B-Tree of order $m$ has the following properties.

- The root is either a leaf or has at least two children.
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and $m$ children.
- All leaves are at the same level in the tree, so the tree is always height balanced.

A B-Tree node is usually selected to match the size of a disk block.

A B-Tree node could have hundreds of children.
Search in a B-Tree is a generalization of search in a 2-3 Tree.

1. Perform a binary search on the keys in the current node. If the search key is found, then return the record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.

2. Otherwise, follow the proper branch and repeat the process.
**B⁺-Trees**

The most commonly implemented form of the B-Tree is the B⁺-Tree.

Internal nodes of the B⁺-Tree do not store records – only key values to guide the search.

Leaf nodes store records or pointers to records.

A leaf node may store more or less records than an internal node stores keys.
B⁺-Tree Insertion

(a) 10 12 23 33 48
(b) 10 12 23 33 48 50
(c) 18 33 48
10 12 15 18 20 21 23 31 33 45 47 48 50 52
(d) 10 12 15 18 20 21 23 30 31 33 45 47 48 50 52
B⁺-Tree Deletion

10 12 15 19 20 21 22 23 30 31 33 45 47 48 50 52

Delete of 19 from original example: Borrow from sibling.
B-Tree Space Analysis

B⁺-Tree nodes are always at least half full.

The B*-Tree splits two pages for three, and combines three pages into two. In this way, nodes are always 2/3 full.

Asymptotic cost of search, insertion and deletion of records from B-Trees, B⁺-Trees and B*-Trees is Θ(log n). (The base of the log is the (average) branching factor of the tree.)

Example: Consider a B⁺-Tree of order 100 with leaf nodes containing 100 records.

1 level B⁺-Tree:

2 level B⁺-Tree:

3 level B⁺-Tree:

4 level B⁺-Tree:

Ways to reduce the number of disk fetches:
- Keep the upper levels in memory.
- Manage B⁺-Tree pages with a buffer pool.