# Searching and traversal

## Sequential search

**Example 5** *Given* $L = (12, 34, 2, 9, 7, 5)$, *Is* $X = 7$ *in the list?*
*We compare each element in the list with the key '7', from left to right.*

```
12 34 2 9 7 5
7
12 34 2 9 7 5
   7
12 34 2 9 7 5
      7
12 34 2 9 7 5
        7
12 34 2 9 7 5
          7        found!
```



## Searching algorithms

These are the algorithms that find a particular data item (called the 'key') in a large collection of such items.

- Sequential search
- Binary search
- Searching on linked lists
- Search trees
- Heaps
- Search on graphs



## Algorithm

Searching in an array A[]:

```
1 read TARGET; set Marker (index) i to 1;
2 while (A[i] != TARGET && not end the list)
3    i++
4 if A[i] == TARGET return i
  else throw an exception 'not found'
```

### Complexity analysis

- Worst case: $N$ comparisons, $O(N)$
- Average case: $(N+1)/2$ comparisons, $O(N)$

# Binary search

**Example 6** $L = (2, 5, 7, 9, 12, 34)$, $X = 5$.

```
2  5  7  9  12  34
      5
2  5  7  9  12  34
   5                    found!
```



# Binary search algorithm 1

A list of data is stored in an array A[0..(A.length-1)]. We use three variables of index, BOT, MID and TOP to mark the sublist each time.

```
1 set BOT=0; TOP=A.length-1;
2 while BOT<TOP
3    set MID=(BOT+TOP) div 2;
4    if A[MID] < TARGET
        set BOT=MID+1;
5    else
        TOP=MID;
6 if TOP=-1 throw an exception 'list is empty';
7 if A[TOP]==TARGET
     return TOP and 'found!';
8 else thow an exception 'not found!'.
```



# Binary search algorithm 2

A list of data is stored in an array A[0..(A.length-1)]. We use three variables, BOT, MID and TOP to mark the sublist each time.

```
1 set BOT=0; TOP=A.length-1;
2 while BOT<TOP
3    set MID=(BOT+TOP) div 2;
4    if A[MID] = TARGET
        return MID and 'found!';
5    else if A[MID]<TARGET
             set BOT=MID+1
6         else TOP=MID-1;
7 throw an exception 'list is empty';
```

**Complexity analysis** Worst case:

$O(\log N)$

# Variations on linked list structures

- Circular linked lists
- Doubly linked circular lists
- Multilinked lists



# Operations on binary trees

- Create()
- isEmpty()
- addTreeNode()
- deleteTreeNode()
- Traversal()



# Binary search trees

**Example 7** *Binary Search Tree and order property.*

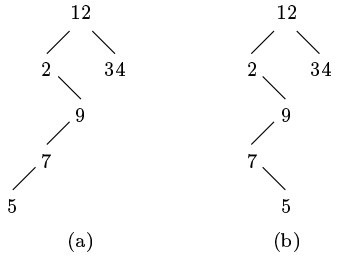


Figure 1: Only (a) is a binary search tree

**Complexity analysis** *Time: O(1)*

# Tree traversals

- PreorderTraversal
- InorderTraversal
- PostorderTraversal

# A binary tree of *chars*

**Example 8** *Given a binary tree, there are 3 ways to traverse all the nodes of the tree.*

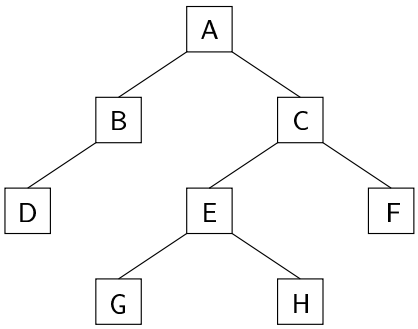


Figure 2: A binary tree



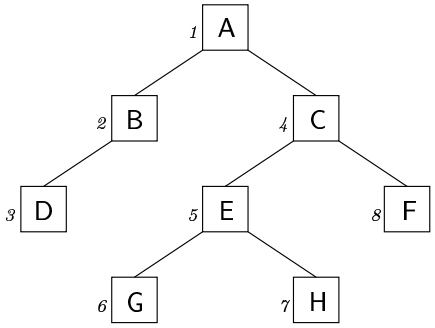


Figure 3: Preorder traversal



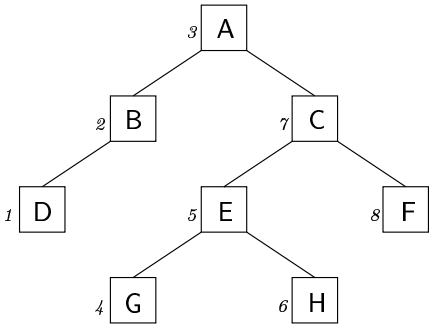


Figure 4: Inorder traversal

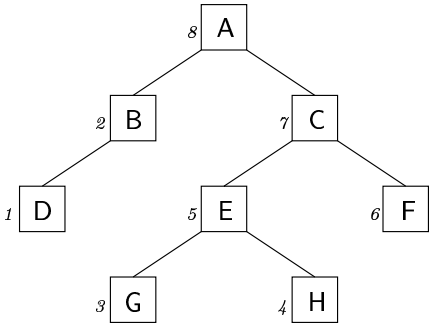


Figure 5: Postorder traversal

# Arithmetic expression tree

The names (Preorder, Inorder and Postorder) come from the three basic forms of arithmetic expressions.

**Example 9** *An arithmetic expression tree, for* $(a+b)^2/(a-b)$,

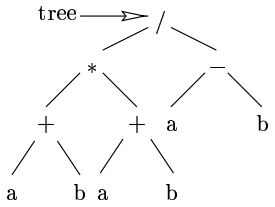


Figure 6: Another expression tree



| Form | Expression |
|---|---|
| Prefix | /*+ab+ab-ab |
| Infix | ((a+b)*(a+b))/(a-b) |
| Postfix | ab+ab+*ab-/ |

# ADT binary trees

```
Create()
isEmpty()
addTreeNode()
deleteTreeNode()
Traversal()

PreorderTrav();
InorderTrav();
PostorderTrav();
```



# BT Implementation using an array

**Example 10** *A complete binary tree*

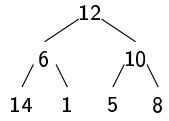


Figure 7:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| A[i] | 12 | 6 | 10 | 14 | 1 | 5 | 8 |

The advantages of array implementation:

A[$i$]'s (if it exists)

- parent: A[$i$ div 2];
- left child: A[$2i$];
- right child: A[$2i + 1$].



**Example 11** *Another complete binary tree*

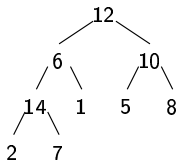


Figure 8:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 12 | 6 | 10 | 14 | 1 | 5 | 8 | 2 | 7 |

**Example 12** *A binary tree*

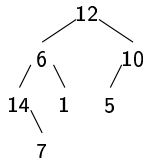


Figure 9:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 12 | 6 | 10 | 14 | 1 | 5 | % | % | 7 |



# BT Implementation using references

Define a binary tree type:

```
public class TreeNode {
  private Object item:
  private TreeNode leftChild;
  private TreeNode rightChild;

  public TreeNode(Object newItem,
     TreeNode lChild, TreeNode rChild) {
    item = newItem;
    leftChild = lChild;
    rightChild = rChild;
  } // Constructor

  public void setItem(Object newItem) {
    item = newItem;
  } // end setItem

  public Object getItem() {
    return item;
```

```
  } // end getItem

  public void setLeft(TreeNode lChild) {
    leftChild = lChild;
  } // end nextNode

  public TreeNode getLeft() {
    return leftChild;
  } // end getNext()

  public void setRight(TreeNode lChild) {
    rightChild = rChild;
  } // end nextNode

  public TreeNode getRight() {
    return rightChild;
  } // end getNext()
} // end TreeNode
```

# Implementation

```
public abstract class KeyedItem {
  private Comparable searchKey;

  public KeyedItem (coparable key) {
    searchKey = key;
  } // end constructor

  public comparable getKey() {
    return searchKey;
  } // end getKey

} // end KeyedItem
```



# ADT binary search tree[a]

- A binary tree with an *order property*
- Suppose all values are distinquished.
  1. A tree node $T.item$ in the BST is greater than all values in its left subtree $T_L$;
  2. $T.item$ is less than all values in its right subtree $T_R$;
  3. Both $T_L$ and $T_R$ are BST.

[a] BST for short



# Heaps - Priority Queues

Heaps and their two properties

**Example 13** *A heap is a complete binary tree (structure property) with a special data arrangement (order property), in which no data in any node bigger (or smaller) than that in its parent.*

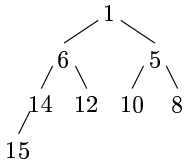


Figure 10: A heap

**Complexity analysis** *Time: O(1)*

# Binary Heaps

**Example 14** *A heap*

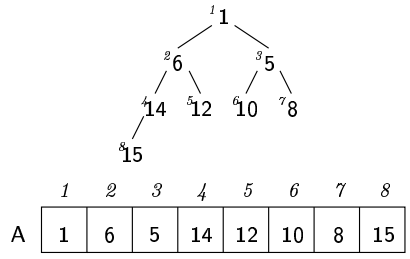


Figure 11: A heap with a $Min$Key root

*or*



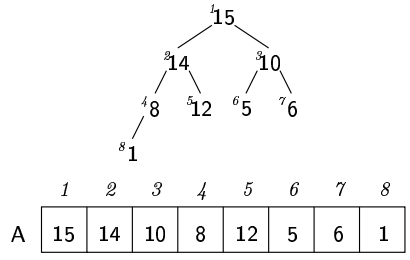


Figure 12: A heap with a $Max$Key root

# Basic heap operations

**AddRoot:** insert an element into the heap as a root and restore the heap properties.

**BuildHeap:** construct the initial heap from a list of items (keys) in arbitrary order.



**Example 15** *Remove the root, move the rightmost leaf to the root.*

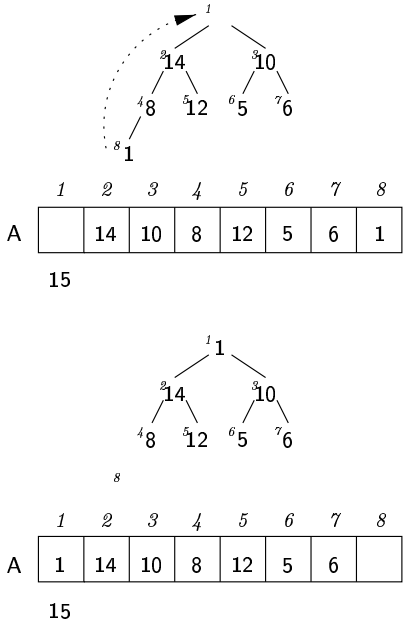


Figure 13: AddRoot-1

*Restore the order property.*

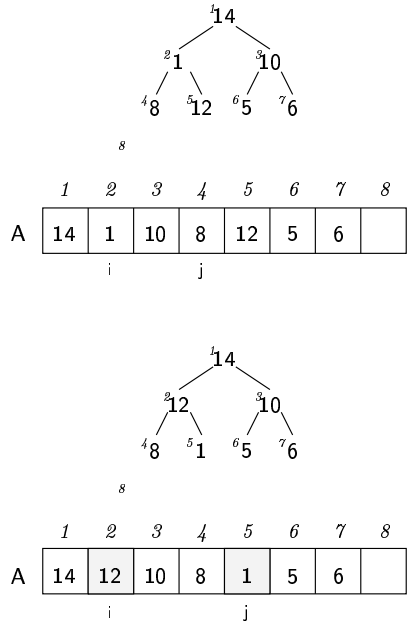


Figure 14: AddRoot-2



# Search on Graphs

# Graphs

**Example 16 A real problem:**

*To connect computers in four buildings by cables.*

*The question: Which pairs of buildings should be directly connected so that the total installation cost would be the minimum?*



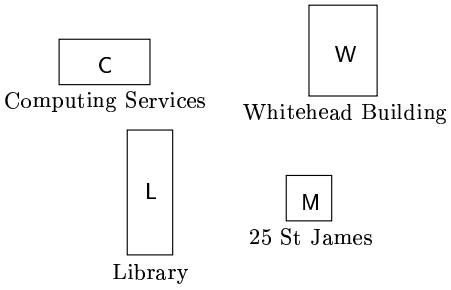


Figure 15:

*The four buildings: labels $C, W, L, M$;*

*The cost between each pair of vertices: lines between vertices.*

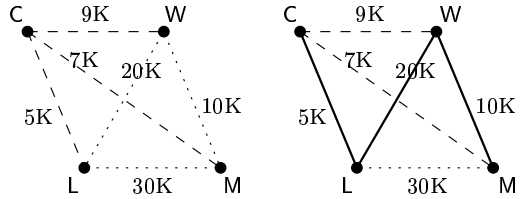


Figure 16:



*All the possible connections:*

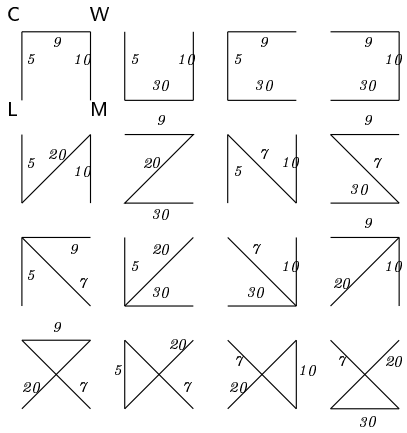


Figure 17: Possible solutions

**Solution** *Direct connections CL, CM and CW.*

*The total cost: 21K.*

**Definition 1** (Graphs) *A Graph $G = (V, E)$ is a finite set of points (vertices) in a space which are interconnected by a finite set of lines (edges), where $V$ is a set of vertices and $E$ is a set of edges.*

**Example 17** *$V = \{C, L, W, M\}$ and $E = \{CL, LW, WM\}$.*

**Example 18** *$V = \{1, 2, 3, 4\}$ and $E = \{12, 23, 34\}$.*

Normally, vertices are labelled by numbers and the edges by letters.



# Two big classes of graphs

- Graphs
- Direct graphs (digraphs)

For graphs, the edge set consists of *non-ordered* pair of vertices, e.g. AB=BA, BC=CB etc.

For directed graphs *digraphs*, the edge set consists of *ordered* pair of vertices, e.g. AB≠BA, etc.

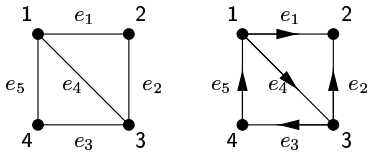


Figure 18: A graph and a digraph

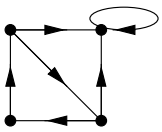

Figure 19: A digraph with a self-loop



# Terms and concepts

**path** A sequence of vertices $v_1, v_2, \ldots, v_k$, where $k \geq 1$ is a path if $(v_i, v_{i+1}) \in V$ for $1 \leq i \leq k$.

**length of a path** The number of edges on the path, which equals $k - 1$, where $k$ is the number of vertices on the path.

**simple path** A path such that all vertices are distinct, except the first and last vertices, i.e. there is no vertices on the path appear more than once.

**loop** A path from a vertex $v$ to itself $(v, v)$. The graphs that we consider here are generally *loopless*.

**cycle** A path $v_1, v_2, \ldots, v_k$ is a cycle if $v_1 = v_k$.

**sparse graph** A graph which has few edges (in a contrast of many vertices).



# Representation of graphs

1. adjacency matrices
2. adjacency lists

The use of different data structures can sometimes improve the efficiency of algorithms.

1. Adjacency matrices

Given a graph $G = (V, E)$, let $n$ be the number of vertices. The adjacency matrix of the graph is a $n \times n$ matrix.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

where

$$a_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

**Note** The adjacency matrix for a undirected graph is always symmetric if row and column nodes are listed in same order.



**Example 19** *For the graph,*

| | C | W | M | L |
|---|---|---|---|---|
| C | 0 | 0 | 0 | 1 |
| W | 0 | 0 | 1 | 1 |
| M | 0 | 1 | 0 | 0 |
| L | 1 | 1 | 0 | 0 |

*For the digraph,*

| | C | W | M | L |
|---|---|---|---|---|
| C | 0 | 0 | 0 | 1 |
| W | 0 | 0 | 1 | 1 |
| M | 0 | 0 | 0 | 0 |
| L | 0 | 0 | 0 | 0 |



2. Adjacency lists

In an adjacency list representation, a graph $G = (V, E)$ is represented by an array of lists, one for each vertex in $V$. For each vertex $u$ in $V$, the list contains all the vertices adjacent to $u$ in an arbitrary order, usually increasing or decreasing order for convenience.

*Why adjacency list?*

It is good for sparse graphs where the number of edges is much less than the square of number of vertices.

**Example 20** *Suppose a digraph with few edges needs to be stored.*

*It is not worth storing the whole matrix in an array because there are many zeros. Adjacency lists can save space in these circumstances:*



| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |

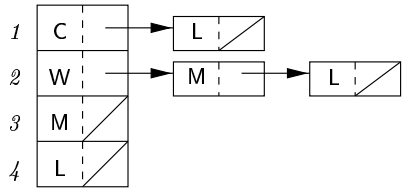


Figure 20: An example of the adjacency list

**Example 21** *This can be implemented by an array:*

| | | |
|---|---|---|
| 1 | C | 5 |
| 2 | W | 6 |
| 3 | M | |
| 4 | L | |
| 5 | L | |
| 6 | M | 7 |
| 7 | L | |

| | | |
|---|---|---|
| 1 | | 5 |
| 2 | | 6 |
| 3 | | |
| 4 | | |
| 5 | 4 | |
| 6 | 3 | 7 |
| 7 | 4 | |

After re-label vertices

Figure 21: An array implementation



# Standard operations

**Create**

**AddEdge**

**RemoveEdge**

**Edge**

**Depth-first Traversal**

**Breadth-first Traversal**

# Depth-first traversal

The main idea of *depth-first traversal* (also called "depth-first search") is to go as far as possible along a path without revisiting any node, then backtrack to the last turning point and go as far as possible down the next path, and so on, until all nodes are visited.

Depth-first search is roughly analogous to preorder traversal of an ordered tree.

Recursive version:

```
method DFS(v:VertexType);
variables
  w:VertexType;

begin
  visit and mark v;
  while there is a unmarked vertex w
       adjacent to v do
    DFS(w)
end
```



**Algorithm** (Depth-first Search)

```
method DFS(adjacencyList:HeaderList;
             v:VertexType);
variables
  S:stack;
  w,x:VertexType;

begin
  Initialise(S);
  visit, mark, and Push(S,v);
  while not Empty(S) do
    begin
      while there a unmarked vertex w
          adjacent to Top(S) do
        visit, mark, and Push(S,w);
      Pop(S,x)
    end {while S ..}
end;
```



# Breadth-first traversal

The *breadth-first traversal* (also called "breadth-first search") is to spiral out from the known to the unknown visiting all places within the same radius before venturing further.

Breadth-first search is roughly analogous to level-by-level traversal of an ordered tree.

## **Algorithm** (Breadth-first Search)

```
method BFS(adjacencyList:HeaderList;
             v:VertexType);
variable
  Q:queue;
  w,x:VertexType;

begin
  Initialize(Q);
  visit and mark v; Enqueue(Q,v);
  while not Empty(Q) do
    begin
      Dequeue(Q,x);
      for each unmarked vertex w adjacent to x do
        begin
          visit and mark w;
          Enqueue(Q,w)
        end {for}
    end {while}
end
```