Fundamentals of Programming

Data structures: tree and heap

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

March 27, 2012

Outline

1. Trees
   - Binary trees
2. AVL tree
3. Heap

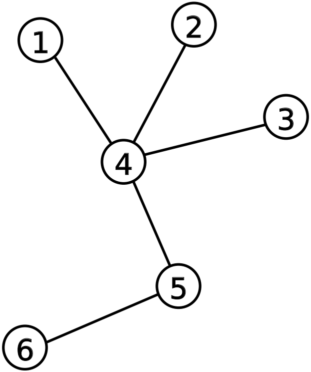
Outline

Representing hierarchies

- One important data structure is the tree
  - In a `List`, nodes are connected with each other in a sequence
  - In a tree, nodes are connected in a hierarchy
- A `Tree` consists of a
  - **root node**,
  - and a set of children nodes,
  - each child node can be the root of a **sub-tree**, or a **leaf node** if it has no children
- A typical example of tree is the organisation of a file system into directories
  - Files are leaf nodes
  - directories are parent nodes

Trees in graph theory

- In graph theory, a tree is a special kind of graph:
  - there must be a simple (unique) path between any two nodes



- Any node can be root!
- This is true for any tree: but picking a node as root, you have a different structure
- Of course, the meaning may change (depending on what is represented)
- A **rooted tree** is a data structure with one specific root
  - here, we are only interested to rooted trees

Representing a tree

- First, we need to represent a node
  - the node contains the data field, plus a list of children nodes

```
struct TreeNode {
    void *pdata;
    LIST children;
};

TREE_NODE *treenode_create(void *data);

typedef struct TreeStruct {
    TreeNode *root;
} TREE;

void tree_init(TREE *t);
...
```

The list contains pointers to TREE_NODE

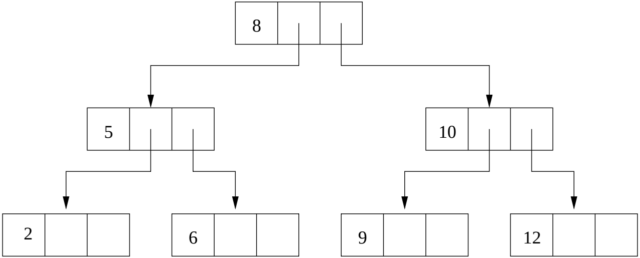
Creates a TREE_NODE

Initialises a TREE

Outline

Binary trees

- A **binary tree** is a data structure where each node can have at most **two children**



- Binary trees are mostly used for
  - Representing binary relationships (i.e. arithmetic expressions, simple languages with binary operators, etc.)
  - Implement search trees

Binary trees definitions

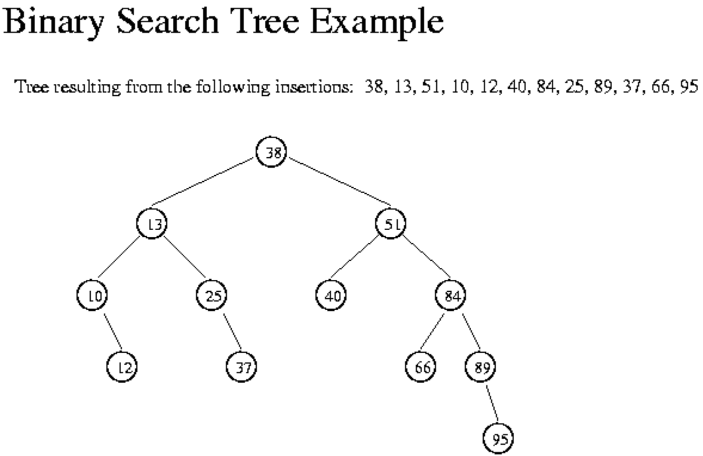
- The **depth** of a node is the length of the path from the root to the node
- The depth (or **height**) of a tree is the length of the path from the root to the deepest node in the tree
- **Siblings** are nodes that share the same parent node
- A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
- A **balanced binary tree** is commonly defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1
  - other definitions are possible, depending on the maximum depth difference we want to allow

Searching in a binary tree

- Given a node that contains element k, the main idea is:
  - insert all elements that are *less than k* to the left
  - insert all other elements to the right
- If the tree is balanced (i.e. it has approximately the same number of nodes in the left and in the right subtrees), searching takes O(log(n))
- Also, insertion takes O(log(n))
  - However, as we insert new elements, the tree may become *unbalanced*

Example of binary tree

- In the following figure we have a tree of integers



Exercises

- Implement a binary tree of integers, without balancement
- Test the algorithm for insertion by printing the tree in-order

Tree of integers

btree-int.h

```
typedef struct btree_node {
    int dato;
    struct btree_node *left;
    struct btree_node *right;
} BNODE;

typedef struct btree_int {
    BNODE *root;
} BTREE_INT;


void btree_init(BTREE_INT *bt);
void btree_insert(BTREE_INT *bt, int d);
int btree_search(BTREE_INT *bt, int dato);
void btree_print_in_order(BTREE_INT *bt);
```

Insertion and search

btree-int.c

```
void __insert(BNODE *node, BNODE *p)
{
    if (p->dato < node->dato) { // to left
        if (node->left == 0) node->left = p;
        else __insert(node->left, p);
    }
    else if (p->dato == node->dato) {
        free(p);
        printf("Element already present!\n");
    }
    else { // to right
        if (node->right == 0) node->right = p;
        else __insert(node->right, p);
    }
}
```

btree-int.c

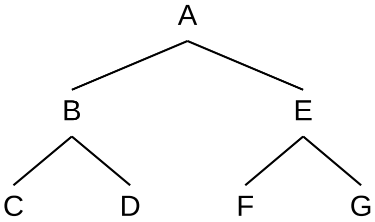
```
int __search(BNODE *node, int dato)
{
    if (node == 0) return 0;

    if (node->dato == dato)
        return 1;
    else if (dato < node->dato)
         return __search(node->left, dato);
    else return __search(node->right, dato);
}
```

Visiting a tree

- There are two ways of listing the contents of a tree
- **Depth-first**
  - Pre-order: first the root node is visited, then the left sub-tree, then the right sub-tree
  - Post-order: first the left sub-tree is visited, then the right sub-tree, then the root node
  - In-order: first the left sub-tree is visited, then the root node, then the right sub-tree
- **Breadth first**
  - First the root node is visited; then all the children; then all the children of the children; and so on

Example



- Breadth first: A B E C D F G
- Pre-order: A B C D E F G
- Post-order: C D B E F G E A
- In-order: C B D A F E G

Visiting in order

btree-int.c

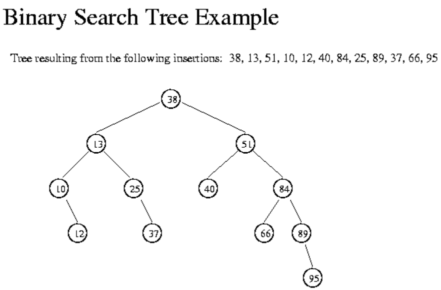
```
void __in_order(BNODE *node)
{
    if (node == 0) return;
    else {
        __in_order(node->left);
        printf("%d, ", node->dato);
        __in_order(node->right);
    }
}
```

- For pre-order and post-order, it is sufficient to change the order in which the print is done
- Is it possible to do it iteratively rather than recursively?

Outline

Height

- The *height* of a tree is how may pointers I have to follow in the worst case before reaching a leaves
- It can be defined recursively;
  - The height of an empty tree is 0
  - The height of a tree is equal to the maximum between the heights of the left and right subtrees plus 1
- Example: what is the height of this subtree?



Balance

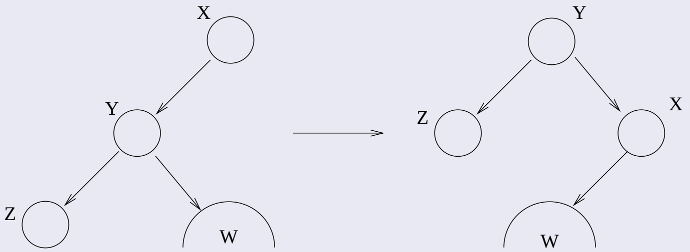
- The difference between the height of the left subtree and the height of the right subtree is called *balance*.
- A tree is said to be *balanced* if
  - the balance is -1, 0 or 1
  - Both the left and the right subtrees are balanced
- (again a recursive definition!)
- Is the tree in the previous slide balanced?
- What is the balance of the tree obtained by example2.txt?

Rotation

- When we insert a new element, the tree can become unbalanced
- Therefore, we have to re-balance it
- The operation that we use to balance the tree must preserve the ordering!
- The balance can be obtained by *rotating a tree*
  - A rotate operation charges the structure of the tree so that the tree becomes balanced after the operation, and the order is preserved
- There are many different implementation of the rotation operation, that produce different types of balanced tree
  - Red-black trees
  - AVL trees
  - etc.
- We will analyze the AVL tree

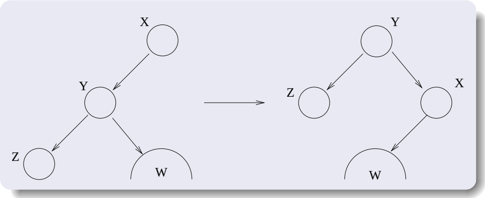
Left-left rotation

- Suppose the tree with root X is unbalanced to the left (i.e. balance $= -2$)
  - In this case, the height of the left subtree (with root Y) is larger than the height of the right subtree by 2 levels
- Also, suppose that the left subtree of Y (which has root Z) is higher than its right subtree
- We apply a *left rotation*:



Left-left rotation

- What happened?
  - Before the rotation,
    - suppose that the right subtree of X had height h,
    - Y had height $h + 2$
    - Z had height $h + 1$
    - W had height h

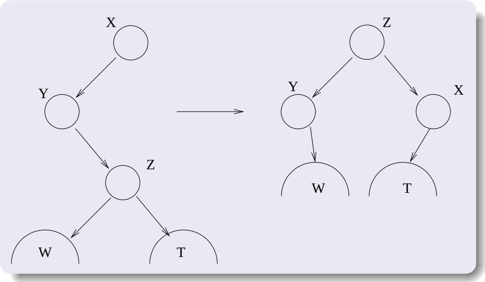


- After the rotation, Y is the new root
  - X has height $h + 1$,
  - Z has height $h + 1$
- Also, notice that the order is preserved:
  - Before the rotation, $Z < Y < W < X$
  - After the rotation, $Z < Y < W < X$

Left-right

- A different case is when the left subtree has balance +1
- In such a case we need to perform a left-right rotation

- Before the rotation,
  - suppose that the right subtree of X had height h,
  - Y had height $h + 2$
  - Z had height $h + 1$
  - W had height h



- After the rotation, Y is the new root
  - X has height $h + 1$,
  - Z has height $h + 1$
- The order is still preserved

Rotations

- There are 4 possible rotations
  - **left-left**: when the tree is unbalanced to the left and the left subtree has balance -1
  - **left-right**: when the tree is unbalanced to the left, and the left subtree has balance +1
  - **right-left**: when the tree is unbalanced to the right, and the right subtree has balance -1
  - **right-left**: when the tree is unbalanced to the right, and the right subtree has balance +1

Rotations

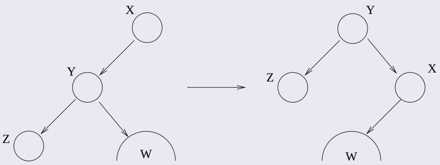


Figure: left-left

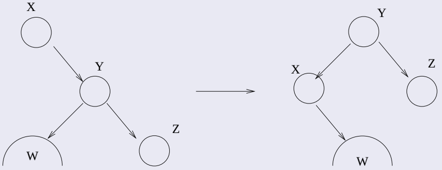


Figure: right-right

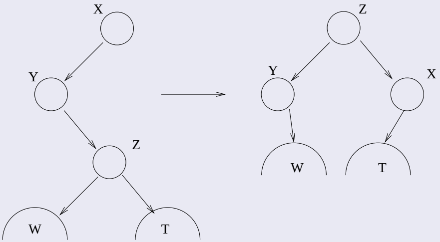


Figure: left-right

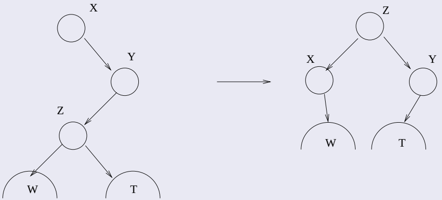


Figure: right-left

Outline

1. Trees
   - Binary trees
2. AVL tree
3. Heap

Heap

- An heap is a data structure that is used mainly for implementing priority queues
- A heap is a binary tree in which, for each node A, the value stored in the node is always greater than the values stored in the childen
- The data structure is also called *max-heap* (or *min-heap* if we require that the node be less than its children)

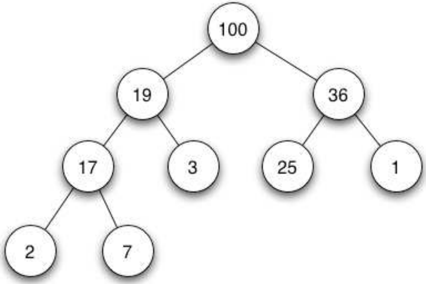


Figure: Example of max-heap

Properties

- Another property of *max-heap* is the fact that the heap is “full” in all its levels except maybe the last one
- Also, on the last level, all nodes are present from left to rightm without holes

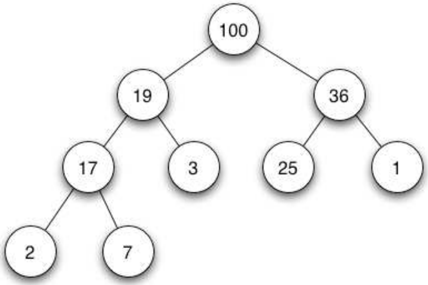


Figure: All nodes are full from left to right

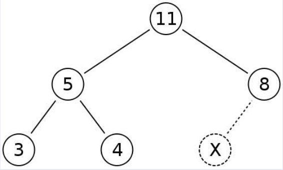
Operations

- The most important operations you can do on a heap are:
  - Insert an element in a ordered fashion
  - Read the top element
  - Extract the top element
- An heap is used mainly for sorted data structures in which you need to quickly know the maximum element

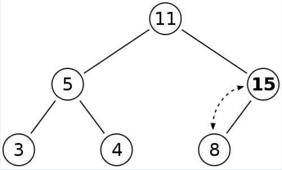
Insertion

- To insert an element, we proceed in two steps
  - First the element is inserted in the first free position in the tree
  - Then, by using a procedure called *heapify*, the node is moved to its correct position by swapping elements
- Suppose we want to insert element 15 in the heap below

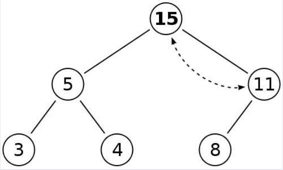
Step 1



Step 2



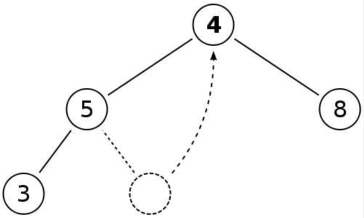
Step 3



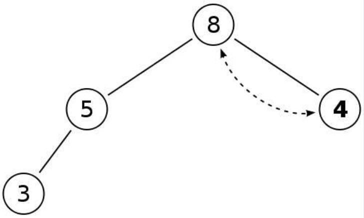
Deleting

- For deleting an element, we proceed in a similar way
  - We first remove the top most element, and we substitute it with the last element in the heap
  - Then, we move down the element to its correct position by a sequence of swaps
- Suppose that we remove the top element in the heap below. We substitute it with the last element (4)

Step 1



Step 2



Heap implementation

- The heap can be efficiently implemented with an array
- The root node is stored at index 0 of the array
- Given a node at index i:
  - its left child can be stored at $2i + 1$
  - its right child can be stored at $2i + 2$
  - the parent of node j is at $\left\lfloor \frac{j-1}{2} \right\rfloor$

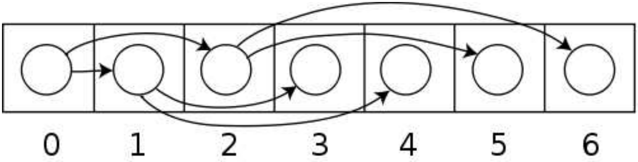


Figure: Efficently storing a heap in an array