Introduction to the C programming language

Lists and Trees

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

Scuola Superiore Sant'Anna – Pisa

March 16, 2010

Outline

1. Searching
2. Lists
3. Balanced Binary Trees
4. AVL tree
5. Heap

Searching

- Suppose we have an address list.
  - For each person name, we have the address and the telephone number.
  - All entries are stored in an array.

Class Entry

- The following class represents an entry

address.hpp

```
class Entry {
    char name[50];
    char address[100];
    char telephone[20];
public:
    Entry();
    Entry(char *s, char *a, char *t);
    char *get_name();
    char *get_address();
    char *get_telephone();
    void print();
};
```

Class AddressBook

- The following class represents an address book with maximum 100 entries

address.hpp

```
class AddressBook {
    Entry array[100];
    int num;
public:
    AddressBook();
    void insert(Entry e);
    Entry search(char *name);
    void printall();
};
```

Implementation of Entry

address.cpp

```
Entry::Entry()
{
    strcpy(name, "");
    strcpy(address, "");
    strcpy(telephone, "");
}

Entry::Entry(char *s, char *a, char *t)
{
    strncpy(name, s, 50);
    strncpy(address, a, 100);
    strncpy(telephone, t, 20);
}
```

Implementation of AddressBook

address.cpp

```
AddressBook::AddressBook() : num(0)
{}

void AddressBook::insert(Entry e)
{
    array[num++] = e;
}

Entry AddressBook::search(char *name)
{
    int i;
    Entry null_entry;
    for (i=0; i<num; i++) {
        if (strcmp(name, array[i].get_name()) == 0)
            return array[i];
    }
    return null_entry;
}
```

- Notice that we must go through the entire list if we want to search for an element

Main

- Reading from file

main1.cpp

```
int main(int argc, char *argv[])
{
    if (argc < 2) {
        cout << "Usage: " << argv[0] << " <filename> " << endl;
        exit(-1);
    }
    ifstream f(argv[1]);
    char s[50]; char a[100]; char t[20];

    while (!f.eof()) {
        f >> s;
        if (f.eof()) break;
        f.getline(a, 99);
        f.getline(t, 19);
        Entry e(s, a, t);
        abook.insert(e);
    }
    abook.printall();
```

Main - II

- Searching names:

main1.cpp

```
    bool quit = false;
    while (!quit) {
        cout << "Insert Name to search: ";
        cin >> s;
        if (strcmp(s, "quit") == 0) break;
        else {
            Entry e = abook.search(s);
            cout << "Result: " << endl;
            e.print();
        }
    }
}
```

Improving the data structures

- We have two problems here:
  - Fixed size: we can allow only 100 entries. It would be better to dynamically change the size of the array depending on the needs of the program
  - Searching takes linear time with the number of entries. Can we do better than that?
- Let’s first solve the second problem

Improving search time

- The idea is to sort the array first
- Then, start looking in the middle
  - If we have found the entry, finish with success
  - If the entry is “greater” than the one we look for, continue looking in the first half
  - If the entry is “less” than the one we look for, continue looking in the second half
- This is a recursive algorithm!
- Exercise:
  - Implement a `sort()` function for the AddressBook class
  - modify the previous “search()” function to implement the algorithm described above (hint: may need an intermediate function)

Lists

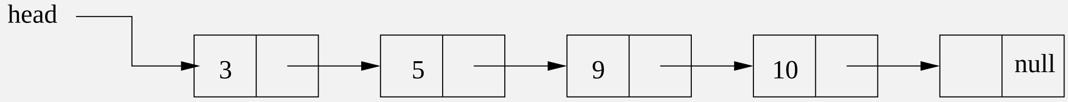
- One important data structure is the linked list
- The nice and important property of a list is the possibility to insert elements at any point without requiring any complex operation

Ordered Insertion

- **Problem:** suppose we have an ordered array of integers, from smalles to largest
- Suppose that we need to insert another number, and that after insertion the array must still be ordered
  - **Solution 1:** Insert at the end, then run a sorting algorithm (i.e. insert sort or bubble sort)
  - **Solution 2:** Identify where the number has to be inserted, and move all successive numbers one position forth
- Both solutions require additional effort to maintain the data structured ordered
- Another solution is to have completely different data structure

Lists

- A list is a chain of linked elements



- Every element of the list contains the data (in this case an integer), and a pointer to the following element in the list

List of Addresses

- We now see how we can use a list to implement an address book
- First of all we define a list element

list.hpp

```
#include "address.hpp"

class ListEntry {
    Entry entry;
    ListEntry *next;
public:
    ListEntry(Entry e);
    void link(ListEntry *next);
    Entry get_data();
    ListEntry *get_next();
};
```

- From `address.hpp`, we reuse the Entry class

List definition

- Now the class AddressList class

list.hpp

```
class AddressList {
    ListEntry *head;
public:
    AddressList();
    void insert(Entry e);
    Entry search(char *s);
    void printall();
};
```

- Notice how similar is the interface with AddressBook

Implementation of ListEntry

list.cpp

```
ListEntry::ListEntry(Entry e): entry(e), next(0)
{}

void ListEntry::link(ListEntry *n)
{
    next = n;
}

Entry ListEntry::get_data()
{
    return entry;
}

ListEntry *ListEntry::get_next()
{
    return next;
}
```

Implementation of AddressList

- The `insert()` operation requires to go through the list until we find the correct position

list.cpp

```
AddressList::AddressList() : head(0)
{}

void AddressList::insert(Entry e)
{
    ListEntry *le = new ListEntry(e);
    ListEntry *p = head;
    ListEntry *q = 0;
    while (p != 0) {
        if (strcmp(p->get_data().get_name(), e.get_name()) > 0) {
            q = p;
            p = p->get_next();
        }
        else break;
    }
    if (q == 0)  // Insertion at the head
        head = le;
    else q->link(le);
    le->link(p);
}
```

Implementation of AddressList

- Searching and printing

list.cpp

```
Entry AddressList::search(char *s)
{
    ListEntry *p = head;
    Entry null_entry;
    while (p != 0) {
        if (strcmp(p->get_data().get_name(), s) == 0)
            return p->get_data();
        else p = p->get_next();
    }
    return null_entry;
}

void AddressList::printall()
{
    ListEntry *p=head;
    while (p != 0) {
        p->get_data().print();
        p=p->get_next();
    }
}
```

Main

- Almost the same as in AddressBook, except for the type of the variable abook, and the includes.

main2.cpp

```
#include "list.hpp"

using namespace std;

AddressList abook;
```

main2.cpp

```
    bool quit = false;
    while (!quit) {
        cout << "Insert Name to search: ";
        cin >> s;
        if (strcmp(s, "quit") == 0) break;
        else {
            Entry e = abook.search(s);
            cout << "Result: " << endl;
            e.print();
        }
    }
```

Problems with lists

- One of the problems with the list is that searching is a O(n) operation
  - while the previous algorithm on the array was O(log(n))
- The list is useful if we frequently insert and extract from the head
  - For example, inside an operating system, the list of processes (executing programs) may be implemented as a list ordered by process priority
  - In general, when most of the operations are inserting/estracting from the headm the list is the simplest and most effective solution

Data structures so far

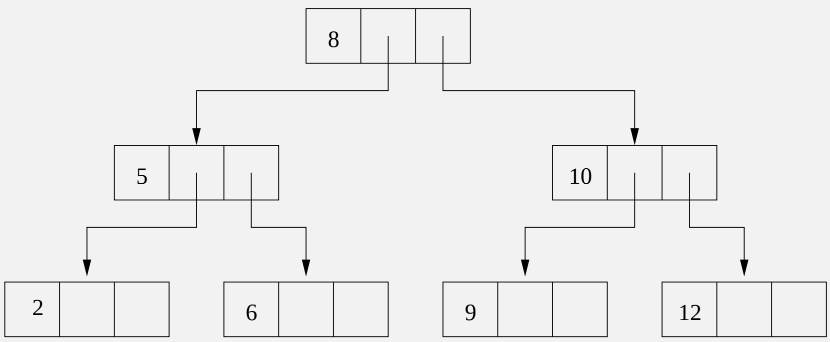
- **Stack**
  - Insertion/extraction only at/from the top (LIFO)
  - All operations are *O(1)*
- **Queue** (Circular Array)
  - Insertion at tail, extraction from head (FIFO)
  - All operations are *O(1)*
- **Array** (random access)
  - Insertion at any point requires *O(n)*
  - Extraction from any point requires *O(n)*
  - Sorting requires *O(n log(n))*
  - Searching (in sorted array) requires *O(log(n))*
- **List** (ordered)
  - Insertion at any point requires *O(n)*
  - Extraction from any point requires *O(1)*
  - Searching requires *O(n)*

More powerful data structures

- No data structure so far allows:
  - Insertion in *O(log(n))*
  - Searching in *O(log(n))*
- It is important to implement efficienlty such data structures, because in most application you exactly need to seach the data structure very efficiently, and insert/remove efficiently
- On such data structure is the balanced binary tree

Trees

- A tree is a data structure where each element can have two *children*
- The parent element can be the *child* of another higher level element
- The topmost element is called *root*



Recursion

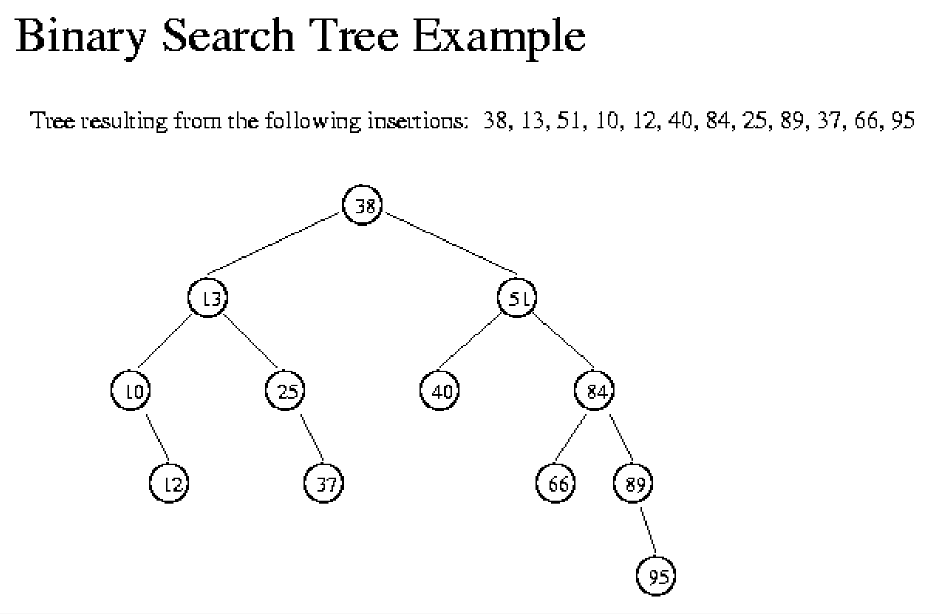
- The tree is a *recursive* data structure
  - The root node has two *subtrees*, one on the left and one on the right
  - Each node can be seen has root of its own subtree
- **Recursive definition**: a tree can be
  - empty (i.e. contains no nodes)
  - consisting of one *root* node, plus one left tree and one right tree
- The tree is defined by itself!

Searching in a tree

- Given a node that contains element *k*, the main idea is:
  - to put all elements that are *less than k* to the left
  - to put all elements that are *greater than k* 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, inserting elements make the tree *unbalanced*

Example of tree

- In the following figure we have a tree of integers



Tree interface

- Here is an example of class that implements a simple tree

simpletree.hpp

```
class AddressTree {
public:
    AddressTree();
    void insert(Entry e);
    Entry search(char *s);
    void print_all();
    void print_structure();
private:
    TreeEntry *root;

    TreeEntry * _insert(TreeEntry *r, Entry e);
    Entry _search(TreeEntry *r, char *s);
    int _get_level(TreeEntry *r);
    void _print_all(TreeEntry *r);
    void _print_level(TreeEntry *r, int l, int n);
};
```

Tree implementation - 1

- The functions insert and search call the internal recursive versions

simpletree.cpp

```
AddressTree::AddressTree() : root(0)
{}

void AddressTree::insert(Entry e)
{
    root = _insert(root, e);
}

Entry AddressTree::search(char *s)
{
    return _search(root, s);
}
```

Tree searching

- Simply looks in the current node, in the left one or in the right one

simpletree.cpp

```
Entry AddressTree::_search(TreeEntry *r, char *s)
{
    Entry null_entry;
    if (r == 0) return null_entry;
    else if (strcmp(r->get_data().get_name(), s) == 0)
        return r->get_data();
    else if (strcmp(r->get_data().get_name(), s) < 0)
        return _search(r->get_left(), s);
    else if (strcmp(r->get_data().get_name(), s) > 0)
        return _search(r->get_right(), s);
    else return null_entry;
}
```

Tree insertion

- Interts to the right or to the left, depending on the ordering

simpletree.cpp

```
TreeEntry *AddressTree::_insert(TreeEntry *r, Entry e)
{
    if (r == 0)
        r = new TreeEntry(e);
    else if (strcmp(r->get_data().get_name(), e.get_name()) < 0)
        r->link_left(_insert(r->get_left(), e));
    else if (strcmp(r->get_data().get_name(), e.get_name()) > 0)
        r->link_right(_insert(r->get_right(), e));
    else if (strcmp(r->get_data().get_name(), e.get_name()) == 0)
        cout << "Element already present" << endl;
    return r;
}
```

The main

- The same as before

maintree.cpp

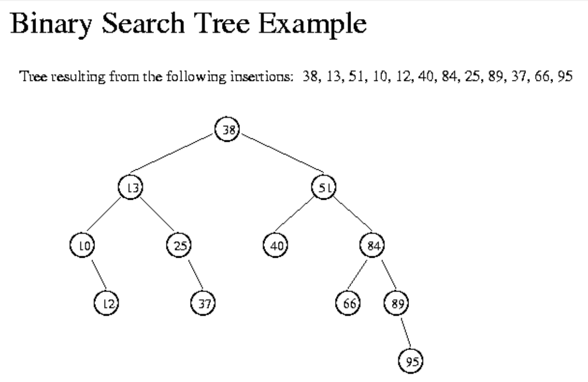
```
AddressTree abook;

int main(int argc, char *argv[])
{
    if (argc < 2) {
        cout << "Usage: " << argv[0] << " <filename> " << endl;
        exit(-1);
    }
    ifstream f(argv[1]);
    char s[50]; char a[100]; char t[20];

    while (!f.eof()) {
        f >> s;
        if (f.eof()) break;
        f.getline(a, 99);
        f.getline(t, 19);
        Entry e(s, a, t);
        abook.insert(e);
    }
    abook.print_all();

    abook.print_structure();

    bool quit = false;
```

Balance

- Unfortunately, the tree is not balances
  - (see output of maintree on example2.txt)
- This means that the insertion and search operation do not necessarily take O(log(n))
  - It is necessary to constantly keep the tree balanced to achieve good performance

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

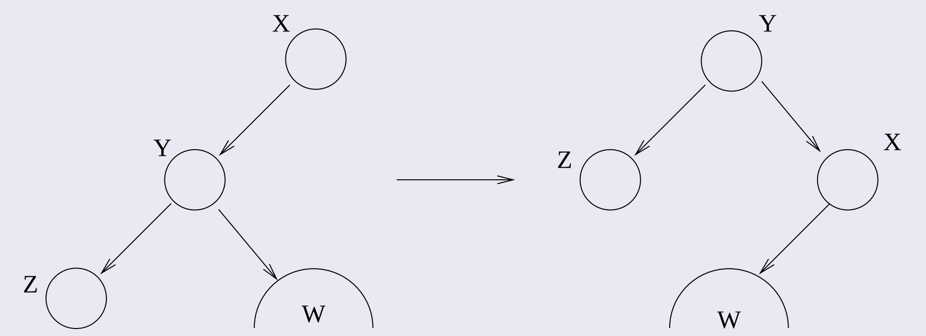
- The difference between the height of the left subtree and the height of the right subtree is called *balance*.
- A tree is said to be *balanced* if
  - the balance is -1, 0 or 1
  - Both the left and the right subtrees are balanced
- (again a recursive definition!)
- Is the tree in the previous slide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

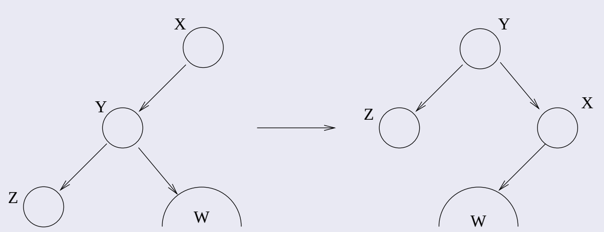
Left-left rotation

- 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

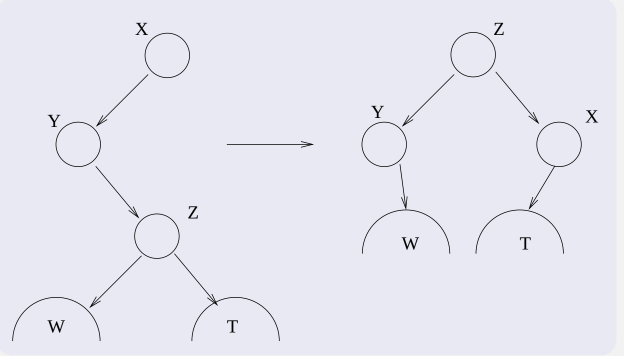


- After the rotation, Y is the new root
  - X has height $h+1$,
  - Z has height $h+1$
- Also, notice that the order is preserved:
  - Before the rotation, $Z < Y < W < X$
  - After the rotation, $Z < Y < W < X$

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

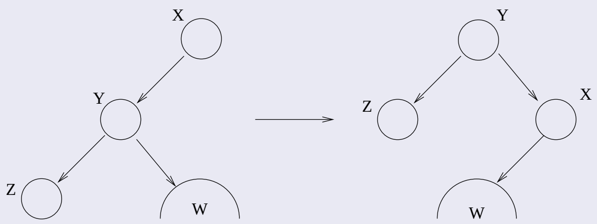


Figure: left-left

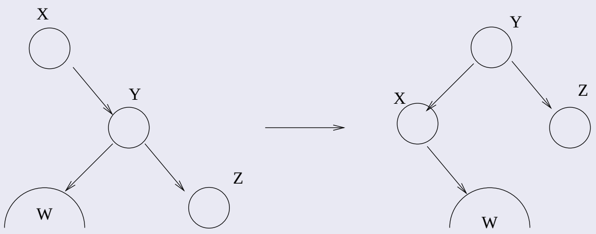


Figure: right-right

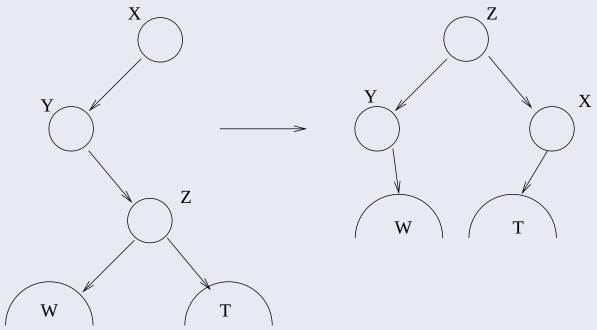


Figure: left-right

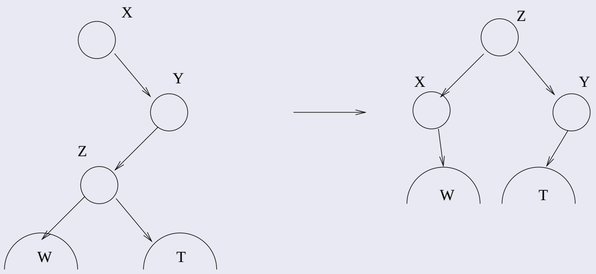


Figure: right-left

Implementation

- Now we look at the implementation

avltree.hpp

```
class AddressTree {
public:
    AddressTree();
    void insert(Entry e);
    Entry search(char *s);
    void print_all();
    void print_structure();
private:
    TreeEntry *root;

    TreeEntry * _insert(TreeEntry *r, Entry e);
    Entry _search(TreeEntry *r, char *s);
    int _get_level(TreeEntry *r);
    void _print_all(TreeEntry *r);
    void _print_level(TreeEntry *r, int l, int n);

    TreeEntry * _rotate_ll(TreeEntry *r);
    TreeEntry * _rotate_lr(TreeEntry *r);
    TreeEntry * _rotate_rl(TreeEntry *r);
    TreeEntry * _rotate_rr(TreeEntry *r);
};
```

Rotations (right)

avltree.cpp

```
TreeEntry * AddressTree::_rotate_rr(TreeEntry *x)
{
    TreeEntry *y = x->get_right();

    x->link_right(y->get_left());
    y->link_left(x);

    return y;
}

TreeEntry * AddressTree::_rotate_rl(TreeEntry *x)
{
    TreeEntry *y = x->get_right();
    TreeEntry *z = y->get_left();

    x->link_right(z->get_left());
    y->link_left(z->get_right());
    z->link_left(x);
    z->link_right(y);

    return z;
}
```

Rotations (left)

avltree.cpp

```
TreeEntry * AddressTree::_rotate_ll(TreeEntry *x)
{
    TreeEntry *y = x->get_left();

    x->link_left(y->get_right());
    y->link_right(x);

    return y;
}

TreeEntry * AddressTree::_rotate_lr(TreeEntry *x)
{
    TreeEntry *y = x->get_left();
    TreeEntry *z = y->get_right();

    x->link_left(z->get_right());
    y->link_right(z->get_left());
    z->link_right(x);
    z->link_left(y);

    return z;
}
```

Height

- The following function returns the tree level:

avltree.cpp

```
int AddressTree::_get_level(TreeEntry *r)
{
    if (r == 0) return 0;
    else return (1 + max(_get_level(r->get_left()),
                         _get_level(r->get_right())));
}
```

- The search remains the same
- Now we look at the insert

Insertion to the left

avltree.cpp

```
TreeEntry *AddressTree::_insert(TreeEntry *r, Entry e)
{
    if (r == 0)
        r = new TreeEntry(e);
    else if (strcmp(r->get_data().get_name(), e.get_name()) < 0) {
        // insert
        r->link_left(_insert(r->get_left(), e));

        // check balance since I inserted to the left, it can be
        // balanced, or in LL or in LR
        int ll = _get_level(r->get_left());
        int rl = _get_level(r->get_right());
        if (ll > (rl + 1)) {
            int lll = _get_level(r->get_left()->get_left());
            int lrl = _get_level(r->get_left()->get_right());

            if (lll > lrl)
                r = _rotate_ll(r);
            else r = _rotate_lr(r);
        }
    }
```

Insertion to the right

avltree.cpp

```
    else if (strcmp(r->get_data().get_name(), e.get_name()) > 0) {
        r->link_right(_insert(r->get_right(), e));

        int ll = _get_level(r->get_left());
        int rl = _get_level(r->get_right());
        if (rl > (ll + 1)) {
            int rrl = _get_level(r->get_right()->get_right());
            int rll = _get_level(r->get_right()->get_left());
            if (rrl > rll) r = _rotate_rr(r);
            else r = _rotate_rl(r);
        }
    }
    else if (strcmp(r->get_data().get_name(), e.get_name()) == 0)
        cout << "Element already present" << endl;

    return r;
}
```

A complete example

- A complete example can be found in program `examples/maintree.cpp`
- **Exercise:** modify the code to change the order in which the elements are stored
- **Exercise:** Modify the code so that:
  1. All elements are stored in an array (i.e. a AddressBook data structure), and only the pointers to the data elements are stored in the tree
  2. Write a different kind of tree that sorts elements by address.
  3. In this way, you will have the same data structure ordered by name and by address at the same time

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)

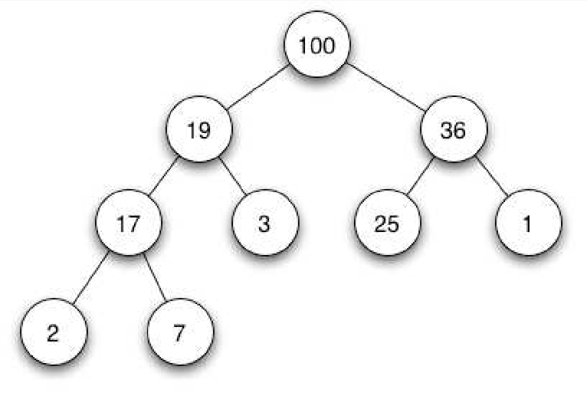


Figure: Example of max-heap

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

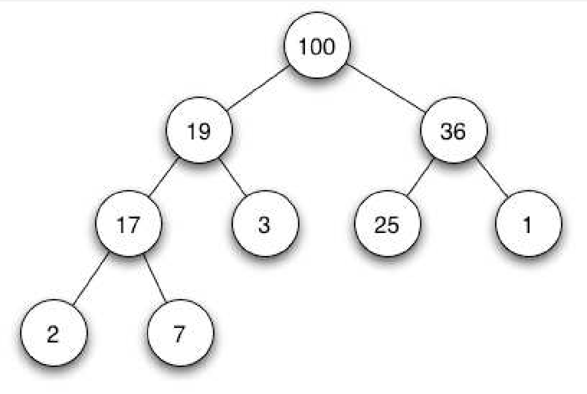


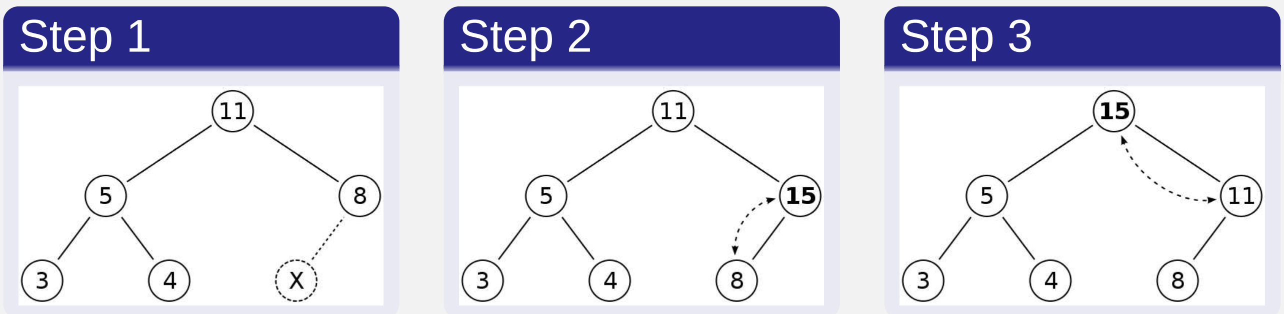
Figure: All nodes are full from left to right

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

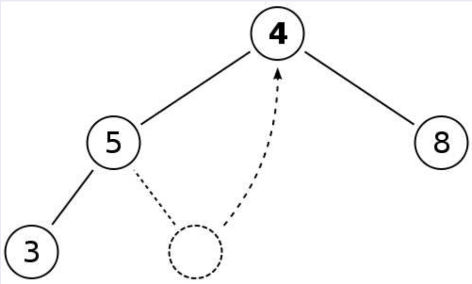
- To insert an element, we proceed in two steps
  - First the element is inserted in the first free position in the tree
  - Then, by using a procedure called *heapify*, the node is moved to its correct position by swapping elements
- Suppose we want to insert element 15 in the heap below



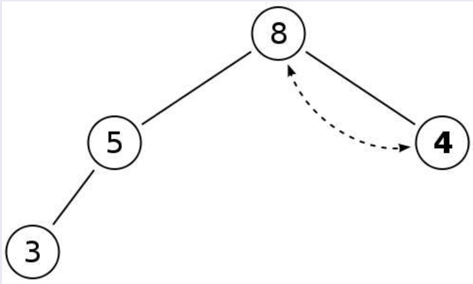
Deleting

- 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 $\left\lfloor \frac{j-1}{2} \right\rfloor$

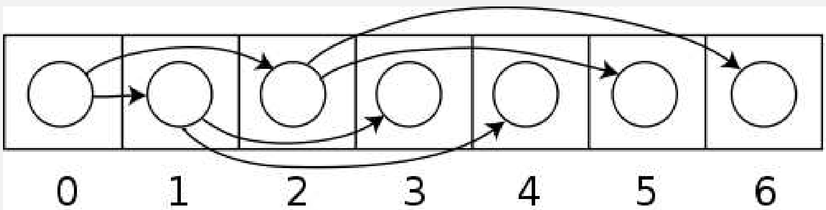


Figure: Efficently storing a heap in an array