COS397 week 6 Recursion revisited

Compute a square root.

To find the square root of a number (`target`), we can use a `high` and a `low` value (by using the assumption that square root of `target` is between `low` and `high`), then we can approximate the square root of `target`.

```
double Sqrt(double target, double low, double hi)
{
  double eps=0.0000005;
  double mid = (low+hi)/2.0;
  cout << target << " " << low << " " << hi << endl;
  if ((mid*mid > target-eps) && (mid*mid < target+eps)) return mid;
  else if (mid * mid > target) return Sqrt(target, low, mid);
  else return Sqrt(target, mid, hi);
}
```

Tower of Hanoi (also called **Towers of Hanoi**), a game that consists of three pegs A, B and C, and a number *n* of discs of different sizes which can slide onto any peg. The puzzle starts with the discs neatly stacked in order of size on one peg, smallest at the top, thus making a conical shape.
The object of the game is to move the entire stack to another peg, obeying the following rules:

- only one disc may be moved at a time
- no disc may be placed on top of a smaller disc

We assume discs are numbered from 1 (smallest, topmost) to *n* (largest, bottommost). To move *n* discs from peg A to peg B:

1. move $n - 1$ discs from A to C. This leaves disc numbered *n* alone on peg A
2. move disc numbered *n* from A to B
3. move $n - 1$ discs from C to B so they sit on disc numbered *n*

To carry out steps 1 and 3, apply the same algorithm again for $n-1$. The entire procedure is a finite number of steps, since at some point the algorithm will be required for $n = 1$. This step, moving a single disc from peg A to peg B, is trivial.

Exercises:

1. Write a recursive function which calculates the triangular number of a given number. A triangular number is the sum of the preceding numbers, so the triangular number 7 has a value of 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28.
2. Write a function that calculates the factorial of given number recursively.
3. Write a function to find recursively the sum of the elements of an array.
4. In a rectangular grid, there are empty and filled cells (marked by 0 or 1, respectively). Two cells are called neighbors if they share a common side. Given the cell (1,1) is empty, write a program, which inputs the size of the rectangular grid (number of rows and number of columns), then an array of 0's and 1's representing the grid row by row, and outputs the number of cells in the largest area of empty cells, containing (1,1), so that any two cells in this area can be connected by a chain of subsequently neighboring cells.
5. Write a program that inputs a positive integer *n* and outputs the sequence of moves to solve the Tower of Hanoi in a form x −> y, where x and y have to be A, B or C.

Linked Lists

A (**singly) linked list** is a data structure consisting of sequence of **nodes**. Each node contains an **element** and a **link** to the next element.

Implement a queue with a singly linked list: the front element is stored at the first node and the rear element is stored at the last element.

A **linked list** is an algorithm for storing a list of items. It is made of any number of pieces of memory (nodes) and each node contains whatever data you are storing along with a pointer (a link) to another node.

By locating the node referenced by that pointer and then doing the same with the pointer in that new node and so on, you can traverse the entire list.

Because a linked list stores a list of items, it has some similarities to an array. But the two are implemented quite differently. An array is a single piece of memory while a linked list contains as many pieces of memory as there are items in the list.

Some advantages that a linked list has over an array are that you can quickly insert and delete items in a linked list. The linked list is an Abstract Data Type (ADT). A simple implementation for a singly linked *node* list:

```
class node
{public:
  object e;
  node* next;
};

class list
{public:
  node* pbeg;
  list(object E)
  {
   pbeg=new node;
   pbeg->next=NULL;
   pbeg->e=E;
  }
  node* findLast()
  {
   node* p=pbeg;
   if(p->next==NULL) return pbeg;
   do p=p->next;
   while(p->next !=0);
   return p;
  }
  void add(object E)
  {
   node* p=findLast();
   p->next = new node;
   p=p->next;
   p->next=NULL;
   p->e=E;
  }
```

```
  void show()
   {
    node* p=pbeg;
    while(p->next !=NULL)
     {cout << p->e << " ";
      p=p->next;
     }
    cout << p->e << "\n";
    }
  node* find(object x)
   {
    node* p=pbeg;
    while(p->e != x)
    {p=p->next;
     if(p==NULL) break;
    }
    return p;
   }
   void removeAfter(node* p)
   {
    if(p==NULL) return;
    if(p->next==NULL)return;
    node* p1=p->next;
    p->next=p1->next;
    delete p1;
   }
};

int main()
{list L(0); L.add(1); L.add(2);
 L.add(3); L.add(4); L.show();
 L.removeAfter(L.find(3));
 L.show();
}
```

Some other methods that you have to implement in order to form the whole class:

```
insertAfter(node* p, object e);
count();
```

Exercises:
1. Write a class to implement singly linked lists.
2. Write a class to implement doubly linked lists.
3. Implement a stack with a singly linked list: the top element is stored at the first node of the list. The space used is O(*n*) and each operation of the stack takes O(1) time.
4. Implement a queue with a singly linked list: the front element is stored at the first node and the rear element is stored at the last element. The space used is O(*n*) and each operation of the queue takes O(1) time.

Implementation of a list using an array:

```
include<iostream>
using namespace std;

struct node {int d, p;};
int n=0; node t[99];

void show()
{int p=1;
 while(p>0)
 {cout << t[p].d << endl; p=t[p].p;}
}

void create()
{int v, p; cin >> v;
 n=1;t[n].d=v;t[n].p=0;
 while(1)
 {cin>>v; if(v==0)break;
  n++;t[n-1].p=n;t[n].d=v;t[n].p=0;
 }
}

int fp;
void find(int v)
{int p=1;
 while(p>0)
 {if(v==t[p].d) {fp=p; return;}
   p=t[p].p;
 }
}

void removeAfter(int p)
{if(t[p].p==0) return;
 int p1=t[t[p].p].p;
 t[p].p=p1;
}

int main()
{create();
 cout << "n=" << n << endl; show();
 fp=0;find(7); cout << "fp=" << fp << endl;
 removeAfter(fp);show();
}
```

Trees

A **binary tree** is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. Each left and each right pointer point recursively to smaller "subtrees" on either side.

The formal recursive definition is: a **binary tree** is either empty, or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.

A "**binary search tree**" (BST) or "ordered binary tree" is a type of binary tree where the nodes are arranged in order: for each node *T*, all elements in its left subtree are less-or-equal to the value of the node *T*, and all the elements in its right subtree are greater than the value of the node *T*.

A simple implementation for a binary tree:

```
#include<iostream>
using namespace std;

class node
{
 public:
  int e;
  node *L, *R;
};

class tree
{
 private:
  int x,flag;
  node* pw;
 public:
  node* root;
 tree(int E)
  {
    root = new node;
    root->e=E;
    root->L=NULL;
    root->R=NULL;
  }
  void show(node* p)
  {
    if(p) cout << p->e << "\n";
    if(p->L) show(p->L);
    if(p->R) show(p->R);
  }

 node* find(int X)
  {
    pw=NULL;
    x=X;
    flag=0;
    findr(root);
    return pw;
  }
```

```
void findr(node* p)
  {
   if(flag) return;
   if(p)
    if((p->e)==x)
     {pw=p; flag=1; return;}
   if(p->L) findr(p->L);
   if(p->R) findr(p->R);
  }

void addAfter(int x, int vL, int
vR)
  {
    node* p=find(x);
    if(p->L==NULL)
     { p->L=new node;
       p->L->e=vL;
       p->L->L=NULL;
       p->L->R=NULL;
     }
     if(p->R==NULL)
     { p->R=new node;
       p->R->e=vR;
       p->R->L=NULL;
       p->R->R=NULL;
     }
  }
};


int main()
{
 tree t(0);
 t.addAfter(0,1,2);
 t.addAfter(1,3,4);
 t.addAfter(2,5,6);
 t.show(t.root);
 cin.get();
 return 0;
}
```

Some other methods that you have to implement in order to form the whole class:

```
removeAfter(node* p);
count();
```

Exercises:

1. Write a class to implement ADT tree with reverse links child – parent.
2. Write a member function to input a tree by its presentation in a text file.
3. Write a function to compare whether two trees are "the same" or not.
4. Implement ADT tree using static arrays.
5. Assuming integer data in nodes are placed in such a way, that in each subtree the value of subtree's root is greater than all values in the left subsubree and is less than all values in the right subsubtree, write a member function for quick search.

A simple implementation for a binary tree using an array:

```
#include<iostream>
using namespace std;
class node
 {public: int d, L, R;};
class tree
{ node t[99];
  public:
   int n; tree() {n=0;}
 void show(int p, int level)
{ level++;
  for(int i=0;i<=level;i++)
    cout << " ";
  cout << t[p].d << endl;
  if(t[p].L>0)
show(t[p].L,level);
  if(t[p].R>0)
show(t[p].R,level);
  level--;
}
void add(int p, int v, char c)
{
 if(c=='L')
  if(t[p].L==0)
   {n++;t[p].L=n;
    t[n].d=v;t[n].L=0;t[n].R=0;}
 if(c=='R')
  if(t[p].R==0)
   {n++;t[p].R=n;t[n].d=v;
    t[n].L=0;t[n].R=0;}
}
```

```
void create()
{ int v, p;
  char c;
  cin >> v;
  n=1;t[n].d=v;t[n].L=0;t[n].R=0;
  while(1)
  {
    cin>>p; if(p==0)break;
    cin>>v>>c;
    add(p,v,c);
  }
}
int fp;
void find(int p, int v)
{
 if(fp>0) return;
 if(t[p].d==v) {fp=p; return;}
 if(t[p].L>0) find(t[p].L,v);
 if(t[p].R>0) find(t[p].R,v);
}
};
int main()
{ tree t; t.create();
  cout << "n=" << t.n << endl;
  t.show(1,0); t.fp=0;
  int v; cin >> v;
  t.find(1,v);
  cout << "fp=" << t.fp << endl;
}
```

Example run: input:

```
20
1 10 L
1 30 R
2 3 L
2 7 R
3 22 L
3 28 R
5 24 R
0
30
```

Output:

```
n=8
  20
   10
    3
    7
     24
   30
    22
    28
fp=3
```

A simple implementation of a binary search ADT using an array

```
#include<iostream>
using namespace std;
class node
 {public:  int d, L, R;};
class bstree
{
node t[99];
public: int n;  bstree(){n=0;}

void show(int p, int level)
{
  level++;
  for(int i=0;i<=level;i++)
   cout << " ";
  cout << t[p].d << endl;
  if(t[p].L>0)
show(t[p].L,level);
  if(t[p].R>0)
show(t[p].R,level);
  level--;
}
void add(int v)
{int p=1;
 while(1)
 {if(v==t[p].d) return;
  if(v<t[p].d)
   {if(t[p].L==0)
     {n++;t[p].L=n;
      t[n].d=v;t[n].L=0;t[n].R=0;
      return;}
      p=t[p].L; continue;
     }
    if(v>t[p].d)
   {if(t[p].R==0)
     {n++;t[p].R=n;
      t[n].d=v;t[n].L=0;t[n].R=0;
      return;}
     p=t[p].R; continue;
   }
  }
}
```

```
void create()
{
  int v, p;
  char c;
  cin >> v;
  n=1;t[n].d=v;t[n].L=0;t[n].R=0;
  while(1)
  {
    cin>>v; if(v==0)break;
    add(v);
  }
}

int fp;
void bfind(int p, int v)
{
 if((fp>0)||(fp==-1)) return;
 if(p==0) {fp=-1; return;}
 if(v==t[p].d) {fp=p; return;}
 if(v<t[p].d) bfind(t[p].L,v);
 if(v>t[p].d) bfind(t[p].R,v);
}
};

int main()
{
  bstree t;
  cout << "create" << endl;
  t.create();
  cout << "n=" << t.n << endl;
  t.show(1,0);
  t.fp=0;
  int v; cin >> v;
  cout << "bfind" << endl;
  t.bfind(1,v);
  cout << "fp=" << t.fp << endl;
}
```

Example run: input:

```
20
10
30
3
7
22
28
24
0
22
```

Output:

```
create
n=8
  20
   10
    3
     7
   30
    22
     28
      24
bfind
fp=6
```

Exercises:

1. Write a program that creates randomly a linked list with 10 integers, displays it, finds if a given integer belongs to the list, removes it from the list and displays the list again..
2. Write a program that creates randomly a linked list with 10 integers, displays it, finds if a given integer belongs to the list, then inserts another given integer after it and displays the list again.
3. Prepare input files for the `tree_demo.cpp` program.
4. Complete the `bst_demo.cpp` and test it with a set of input files.

STL, C++ Standard Template Library

(http://www.cppreference.com)

Stack

```
#include<stack>
#include<iostream>
using namespace std;

int main()
{
  stack<int> s;
  for( int i=0; i < 10; i++ ) s.push(i);
  while( !s.empty() )
   {
     cout << s.top() << " ";
     s.pop();
   }
}
```

Queue

```
#include<queue>
#include<iostream>
using namespace std;

int main()
{
  queue<int> s;
  for( int i=0; i < 10; i++ ) s.push(i);
  while( !s.empty() )
   {
     cout << s.front() << " ";
     s.pop();
   }

}
```

List and **vector**

Lists are sequences of elements stored in a linked list. Compared to vectors, they allow fast insertions and deletions, but slower random access.

Vectors contain contiguous elements stored as an array. Accessing members of a vector or appending elements can be done in constant time, whereas locating a specific value or inserting elements into the vector takes linear time.

```
#include<list>
#include<iostream>
using namespace std;
int main()
{
  list<int> s;
  for(int i = 0; i < 10; i++) s.push_back(i);
// Display the list
   list<int>::iterator it;
   for(it = s.begin(); it != s.end(); it++) cout << *it << endl;
//remove(val) removes all elements that are equal to val
  s.remove(5);
// Display the list
   for(it = s.begin(); it != s.end(); it++ ) cout << *it << endl;
//find val
 int val=6;
 for(it = s.begin(); it != s.end(); it++) if(*it==val) break;
// insert(loc,val)  inserts val before loc,
//returning an iterator to the element inserted
 s.insert(it,99); // runs in constant time
 // Display the list
 for(it = s.begin(); it != s.end(); it++) cout << *it << endl;
}

#include<vector>
#include<iostream>
using namespace std;
int main()
{vector<int> v;
 for( int i = 0; i < 9; i++ ) v.push_back(i);
 cout << "The first element is " << v.front()
      << " and the last element is " << v.back() << endl;
 vector<int>::iterator it;
 for(it = v.begin(); it != v.end(); it++ ) cout << *it << endl;

 for( int i = 0; i < 9; i++ ) cout << v[i] << endl;

 int val=6;
 for(it = v.begin(); it != v.end(); it++) if(*it==val) break;
// erase(loc)  deletes the element at location loc,
// returning an iterator to the element inserted
  v.erase(it);
 for(it = v.begin(); it != v.end(); it++ ) cout << *it << endl;
 val=7;
 for(it = v.begin(); it != v.end(); it++) if(*it==val) break;
 v.insert(it,99); // runs in linear time
 // Display
 for(it = v.begin(); it != v.end(); it++) cout << *it << endl;
}

iterator erase( iterator loc );
iterator erase( iterator start, iterator end );
```

The erase() function either deletes the element at location *loc*, or deletes the elements between *start* and *end* (including *start* but not including *end*). The return value is the element after the last element erased. The first version of erase (the version that deletes a single element at location *loc*) runs in constant time for *lists* and linear time for *vectors*, and *strings*. The multiple-element version of erase always takes linear time.