Laurea triennale in Informatica

modulo (CFU 6) *di*

Programmazione II e Lab.

prof. Mariarosaria Rizzardi

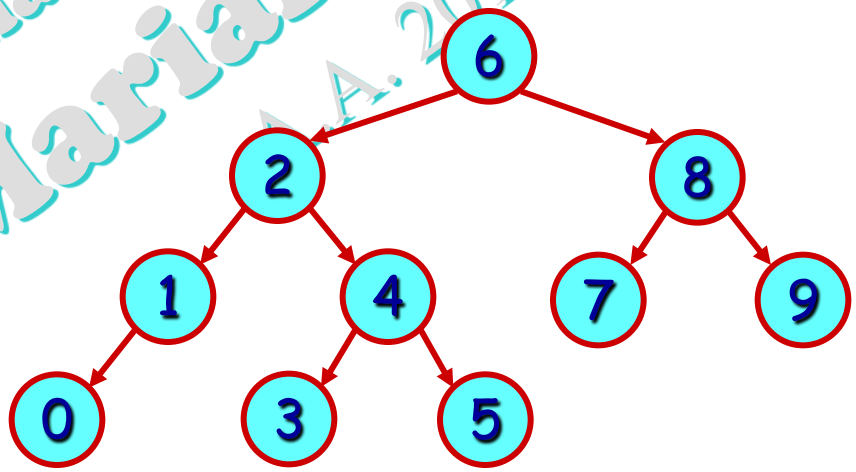
Centro Direzionale di Napoli – Isola C4
stanza: n. 423 – IV piano Lato Nord
tel.: 081 547 6545
email: mariarosaria.rizzardi@uniparthenope.it

Programmazione in C++:

- implementazione di un BST (Binary Search Tree)
- la classe template `priority_queue` di STL

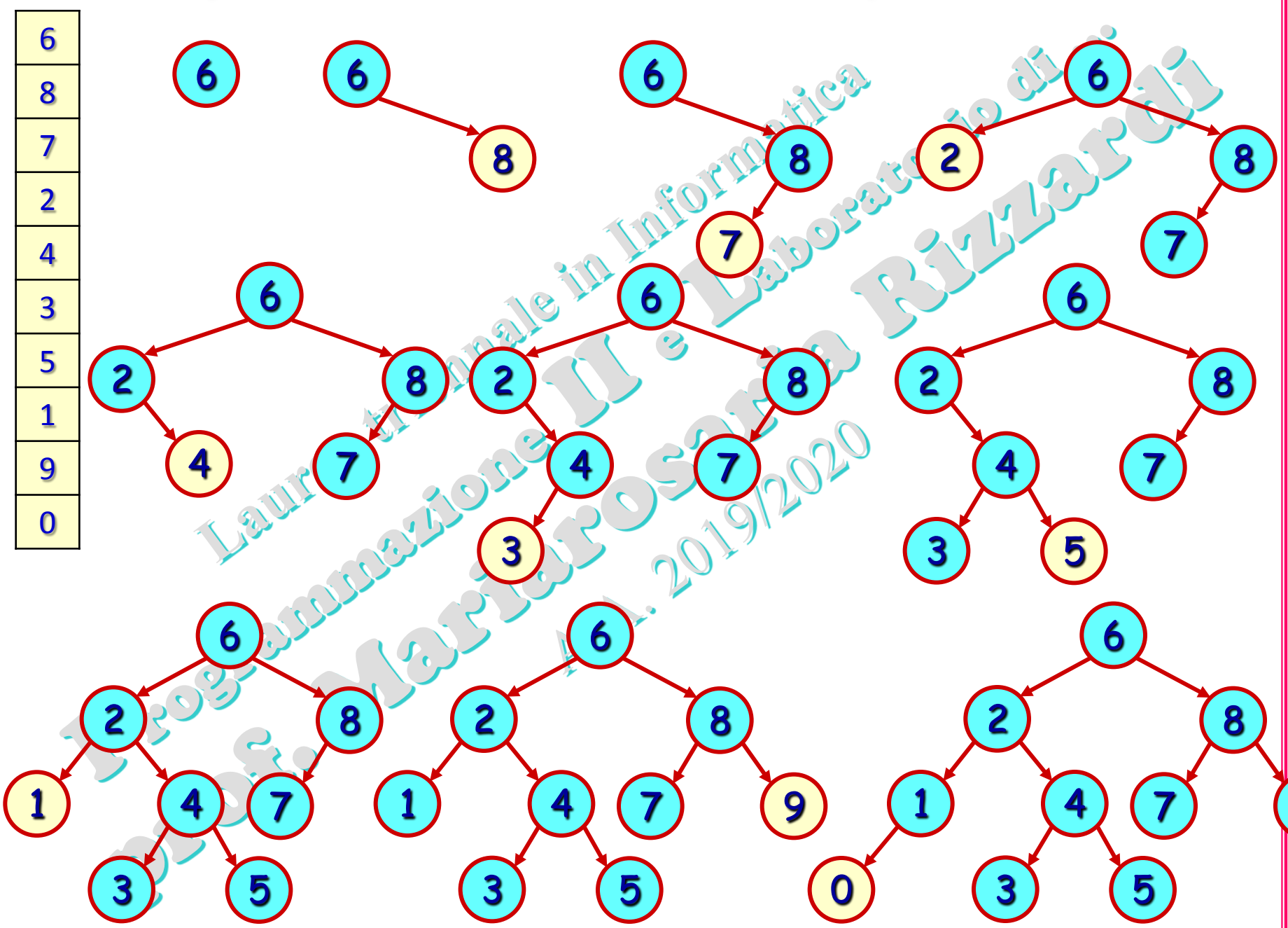
La Libreria Standard del C++ (**STL - Standard Template Library**) non mette a disposizione classi template per implementare la struttura dati albero binario e albero binario di ricerca. È compito del programmatore implementare tali classi (oppure usare librerie specifiche come BOOST, LEDA).

Esempio: Binary Search Tree



Esempio: costruzione di un Binary Search Tree

6
8
7
2
4
3
5
1
9
0



possibili classi per un Binary Search Tree

classe del nodo di un albero binario

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

typedef int KEY_NODO;

class BTREEnode {
    KEY_NODO info;
    BTREEnode *sx, *dx;  // pointers to left and right children
  public:
 costruttore
    BTREEnode(KEY_NODO i=0) : info(i), sx(nullptr), dx(nullptr) {}
    void setInfo(KEY_NODO i) {  this->info=i;  }
    KEY_NODO getInfo()       {  return this->info;  }
    void displayInfo() {  cout << this->getInfo() << endl;  }
    void setSxPtr(BTREEnode *leftPtr)  {  this->sx=leftPtr;  }
    void setDxPtr(BTREEnode *rightPtr) {  this->dx=rightPtr;  }
    BTREEnode *getSxPtr() {  return this->sx;  }
    BTREEnode *getDxPtr() {  return this->dx;  }
};
```

possibile classe per un Binary Search Tree

classe del BST

```
class BSTree {
  private:
    BTREEnode *root;
  public:
    BSTree() : root(nullptr) {} // costruttore default: albero vuoto
    BSTree(KEY_NODO i) {  this->root = new BTREEnode(i);  }
    BSTree(KEY_NODO i, BTREEnode* &pt); // passato per reference
    void deleteBST(BTREEnode* pt); // cancella il sottoalbero di radice pt
    void deleteBST();
    ~BSTree() {  deleteBST();  }
    BSTnode *getBSTroot() {  return this->root;  }
    void displayPreOrder(BTREEnode* pt);
    void displayInOrder(BTREEnode* pt);
    void displayPostOrder(BTREEnode* pt);
    BTREEnode *binarySearch (KEY_NODO i, BTREEnode* pt);
    BTREEnode *addNodeBSTree(KEY_NODO i, BTREEnode* pt);
};
```

distruttori costruttori

alcuni possibili metodi per la classe BSTree

```
BSTree::BSTree(KEY_NODO i, BTREEnode* &pt)
{
    pt = new BTREEnode(i);
}
void BSTree::deleteBST(BTREEnode* pt)
{
    if ( pt == nullptr )  return;
    deleteBST(pt->getSxPtr());
    deleteBST(pt->getDxPtr());
    delete pt;
}
void BSTree::deleteBST()
{
    deleteBST(this->root);
}
void BSTree::displayPreOrder(BTREEnode* pt)
{
    if (pt == nullptr)  return;
    pt->displayInfo(); // visualizza radice
    displayPreOrder(pt->getSxPtr()); // visualizza sottoalbero sinistro
    displayPreOrder(pt->getDxPtr()); // visualizza sottoalbero destro
}
...
```

alcuni possibili metodi per la classe BSTree

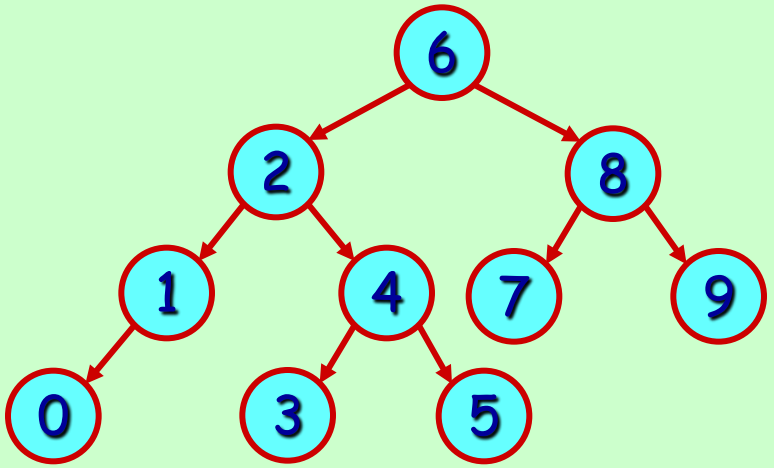
```
BTREEnode* BSTree::binarySearch(KEY_NODO i, BTREEnode* pt)
{
    if ( pt == nullptr )
        return nullptr;      // non trovato
    else
    {
        if ( i == pt->getInfo() )
            return pt;       // trovato
        if ( i < pt->getInfo() )
            return binarySearch(i, pt->getSxPtr());
        else // i > pt->getInfo()
            return binarySearch(i, pt->getDxPtr());
    }
}
```

alcuni possibili metodi per la classe BSTree

```
BTREEnode* BSTree::addNodeBSTree(KEY_NODO i, BTREEnode* pt)
{
    if ( pt == nullptr )
        BSTree(i, pt);
    else // pt != nullptr
        if ( i <= pt->getInfo() )
            if ( pt->getSxPtr() == nullptr )
                pt->setSxPtr(new BTREEnode(i));// inserisce nuovo nodo come figlio sx
            else
                pt->setSxPtr(addNodeBSTree(i, pt->getSxPtr()));
        else // i > pt->getInfo()
            if ( pt->getDxPtr() == nullptr )
                pt->setDxPtr(new BTREEnode(i));// inserisce nuovo nodo come figlio dx
            else
                pt->setDxPtr(addNodeBSTree(i, pt->getDxPtr()));
    return pt;
}
```

Funzione main() per la classe BSTree

```
#include <iostream>
#include "BST.hpp"
using namespace std;
int main()
{
    cout << "Build a Binary Search Tree" << endl;
    BSTree ABR(6); // nodo radice
    BTREEnode *BSTroot = ABR.getBSTroot();
    ABR.addNodeBSTree(8,BSTroot);
    ABR.addNodeBSTree(7,BSTroot);
    ABR.addNodeBSTree(2,BSTroot);
    ABR.addNodeBSTree(4,BSTroot);
    ABR.addNodeBSTree(3,BSTroot);
    ABR.addNodeBSTree(5,BSTroot);
    ABR.addNodeBSTree(1,BSTroot);
    ABR.addNodeBSTree(9,BSTroot);
    ABR.addNodeBSTree(0,BSTroot);
    cout << "\nPre order visit:" << endl;
    ABR.displayPreOrder(ABR.getBSTroot());
    KEY_NODO info = 5;    BTREEnode *pt = ABR.binarySearch(info, ABR.getBSTroot());
    if ( pt == nullptr )
        cout << "\n\ninfo = " << info << ": not found" << endl;
    else
    {  cout<<"\n\ninfo     = "<< info <<": found at address pt = "<< (unsigned long long)pt<<endl;
       cout << "pt->getInfo() = " << pt->getInfo() << endl;
    }
    info = 100;    pt = ABR.binarySearch(info, ABR.getBSTroot());
    if ( pt == nullptr )
        cout << "\n\ninfo = " << info << ": not found" << endl;
    else
    {  cout<<"\n\ninfo     = "<< info <<": found at address pt = "<< (unsigned long long)pt<<endl;
       cout << "pt->getInfo() = " << pt->getInfo() << endl;
    }
    return 0;
}
```



6
8
7
2
4
3
5
1
9
0

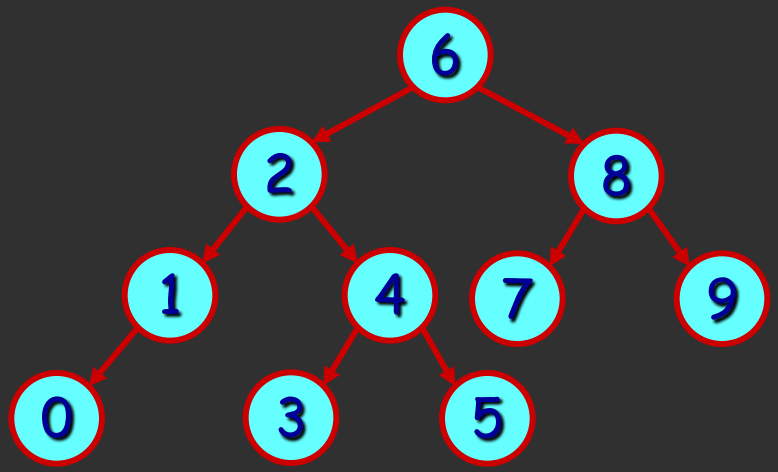
Output

```
Build a Binary Search Tree
Pre order visit:
6
2
1
0
4
3
5
8
7
9

info       = 5: found at address pt = 6118240
pt->getInfo() = 5


info       = 0: found at address pt = 6118336
pt->getInfo() = 0


info = 100: not found
```

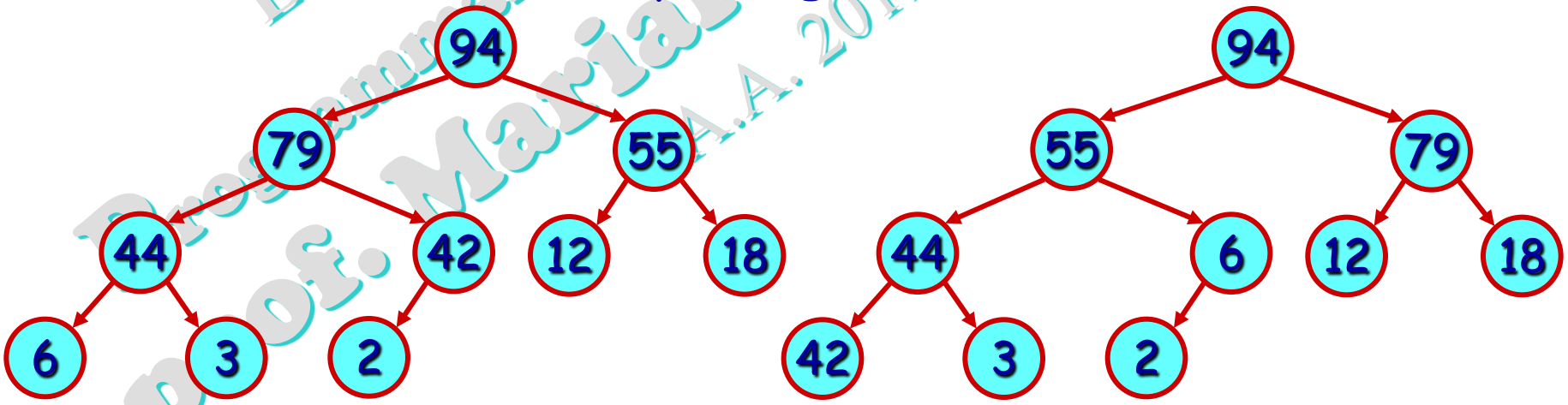


La Libreria Standard del C++ (**STL - Standard Template Library**) mette a disposizione la classe template **priority_queue** per implementare la struttura dati coda con priorità (max-heap).

Esempio: max-heap

Proprietà max-heap: key(nodo) > key(figlio_sx(nodo)) key(figlio_dx(nodo))

Due heap con gli stessi dati

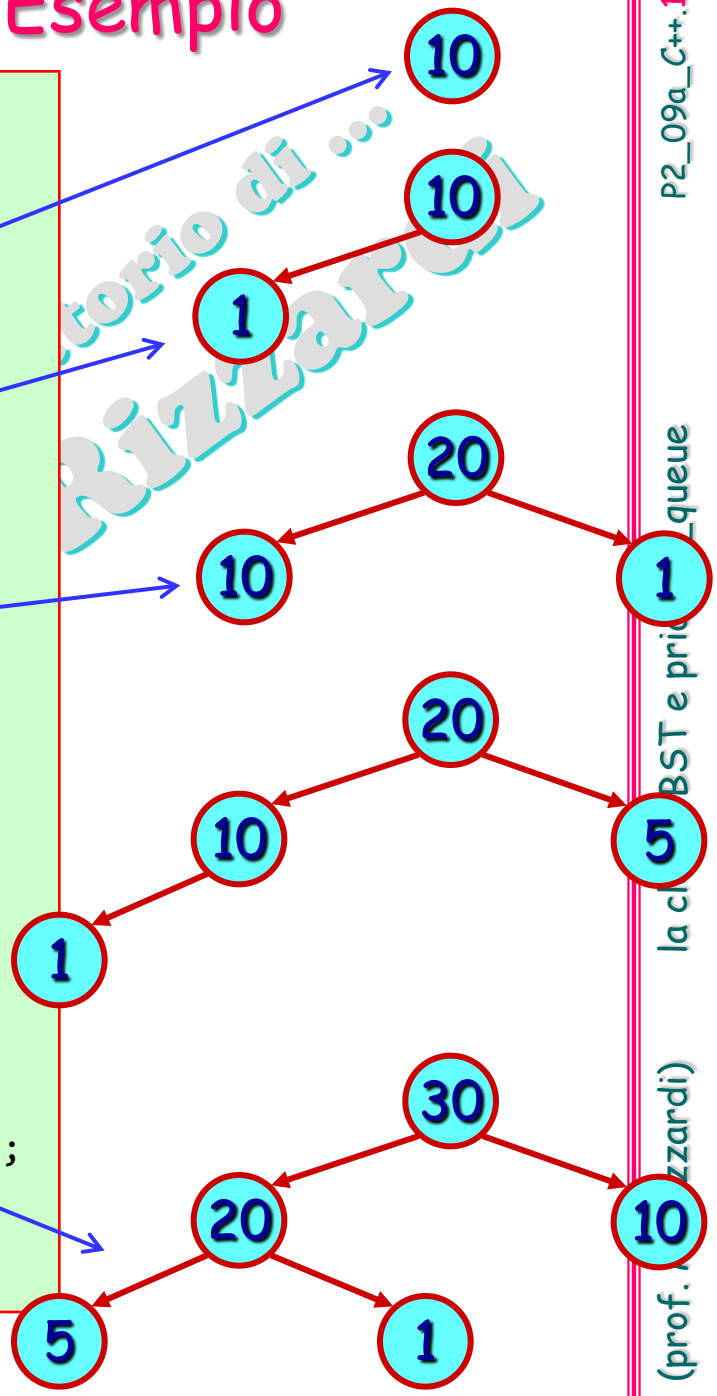


Ricordare che l'heap non è unico!

std::priority_queue<T>: Esempio

```
#include <iostream>
#include <queue>
using namespace std;
void showPriorityQueue(priority_queue <int> pq);
int main() {
    priority_queue <int> pq;
    cout << "\npq.size() = " << pq.size();
    if ( pq.empty() )
        cout << "\tempty priority queue\n";
    else
        cout << "\tnot empty priority queue\n";
    pq.push(10);
    cout << "pq.push(10) ==> "; showPriorityQueue(pq);
    pq.push(1);
    cout << "pq.push(1)  ==> "; showPriorityQueue(pq);
    pq.push(20);
    cout << "pq.push(20) ==> "; showPriorityQueue(pq);
    pq.push(5);
    cout << "pq.push(5)  ==> "; showPriorityQueue(pq);
    pq.push(30);
    cout << "pq.push(30) ==> "; showPriorityQueue(pq);

    cout << "\npq.size() = " << pq.size();
    if ( !pq.empty() )
    {   cout << "\npq.top()  = " << pq.top();
        pq.pop();
        cout << "\npq.pop()  ==> "; showPriorityQueue(pq);
    }
    return 0;
}
```



std::priority_queue<T>: Esempio (cont.)

```
void showPriorityQueue(priority_queue <int> pq)
{
    while ( !pq.empty() )
    {
        cout << '\t' << pq.top();
        pq.pop();
    }
    cout << '\n';
}
```

il metodo top() visualizza l'elemento in cima all'heap (la radice dell'albero binario)

Perché, nonostante venga usato il metodo pop(), la coda con priorità incrementa comunque il suo contenuto?

```
pq.size() = 0   empty priority queue
pq.push(10) ==> priority queue contains:   10
pq.push(1)  ==> priority queue contains:   10    1
pq.push(20) ==> priority queue contains:   20   10    1
pq.push(5)  ==> priority queue contains:   20   10    5    1
pq.push(30) ==> priority queue contains:   30   20   10    5    1

pq.size() = 5
pq.top()  = 30
pq.pop() ==> priority queue contains:   20   10    5    1
```

http://www.cplusplus.com/reference/queue/priority_queue/

std::priority_queue<T> (max heap)

sintassi generale

```
template <class T,
          class Container = vector<T>,
          class Compare = less<typename Container::value_type>
         > class priority_queue;
```

per un min heap

```
typedef priority_queue < int,
                         vector<int>,
                         greater<int>
                       > minHeapPriorityQueue;

int main() {
     minHeapPriorityQueue pq;
...
}
```

sostituisce std::less

```
template <class T> struct less {
  bool operator() (const T& x, const T& y) const { return x<y; }
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};
```

```
pq.size() = 0   empty priority queue
pq.push(10) ==> priority queue contains:   10
pq.push(1)  ==> priority queue contains:   1    10
pq.push(20) ==> priority queue contains:   1    10   20
pq.push(5)  ==> priority queue contains:   1    5    10   20
pq.push(30) ==> priority queue contains:   1    5    10   20   30

pq.size() = 5
pq.top()  = 1
pq.pop() ==> priority queue contains:   5       10       20       30
```