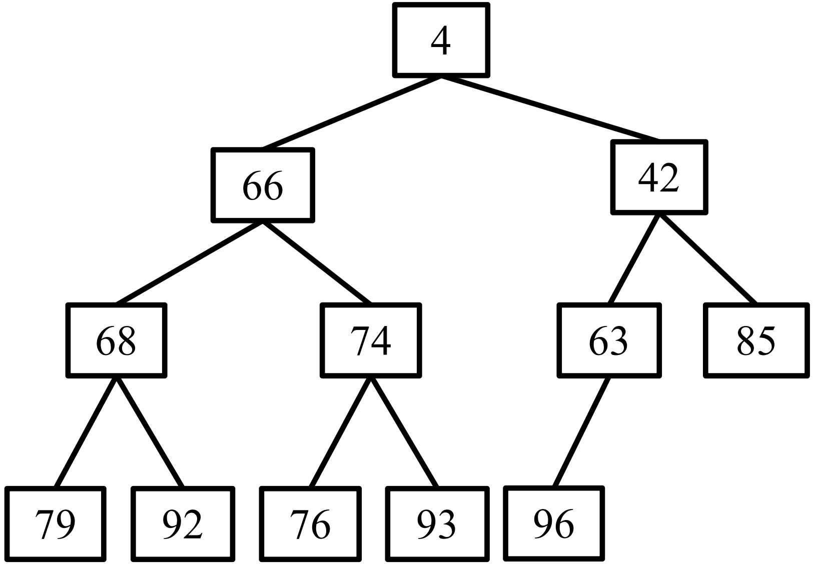
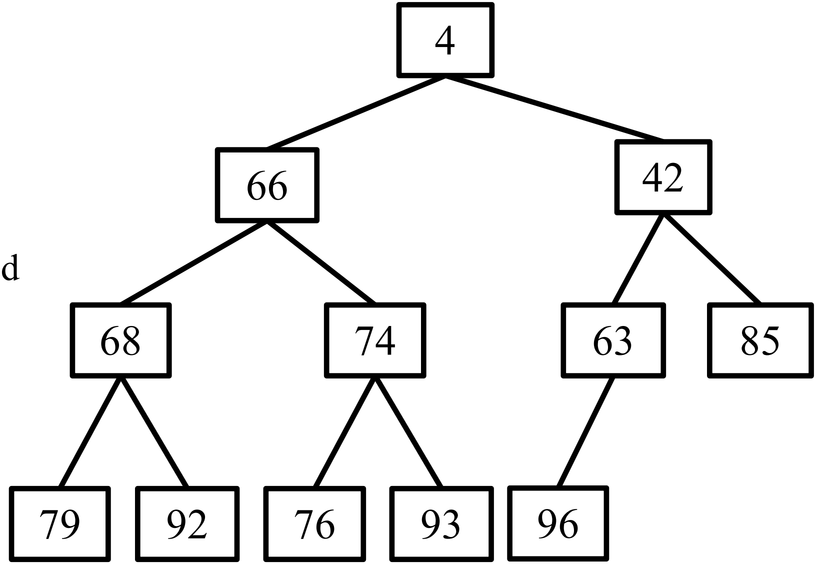
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation



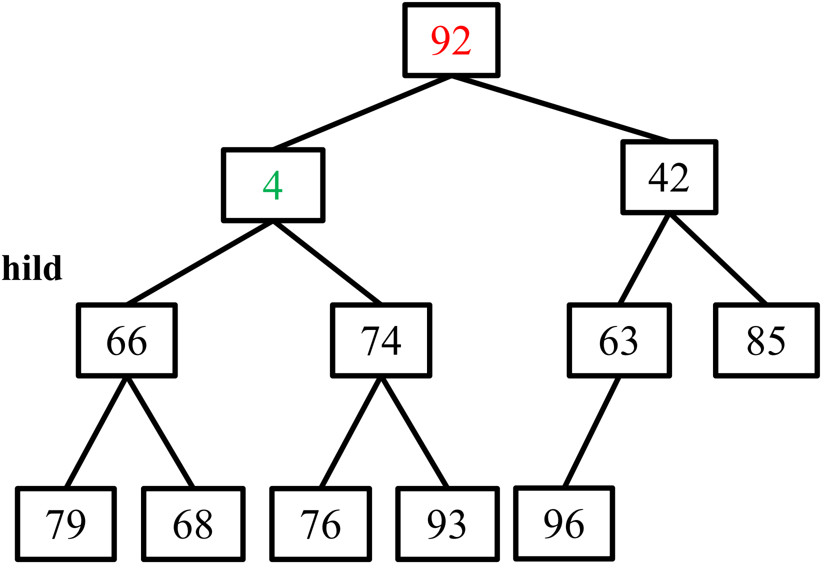
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent



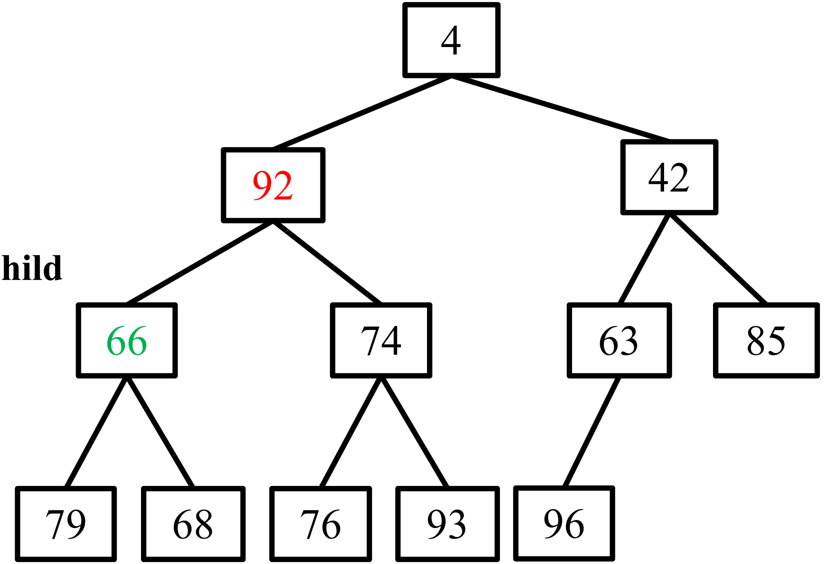
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - **if child is prior, swap with prior child**
  - *siftup*
    - if prior to parent, swap with parent



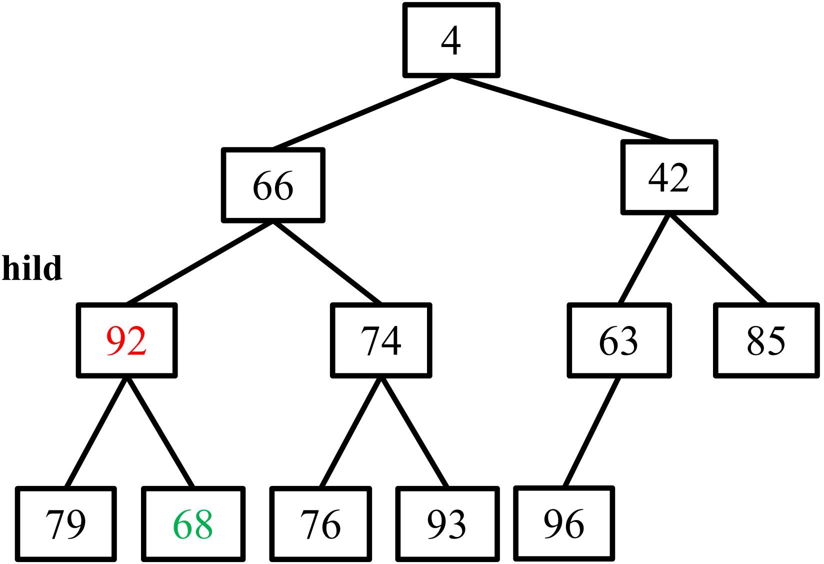
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - **if child is prior, swap with prior child**
  - *siftup*
    - if prior to parent, swap with parent



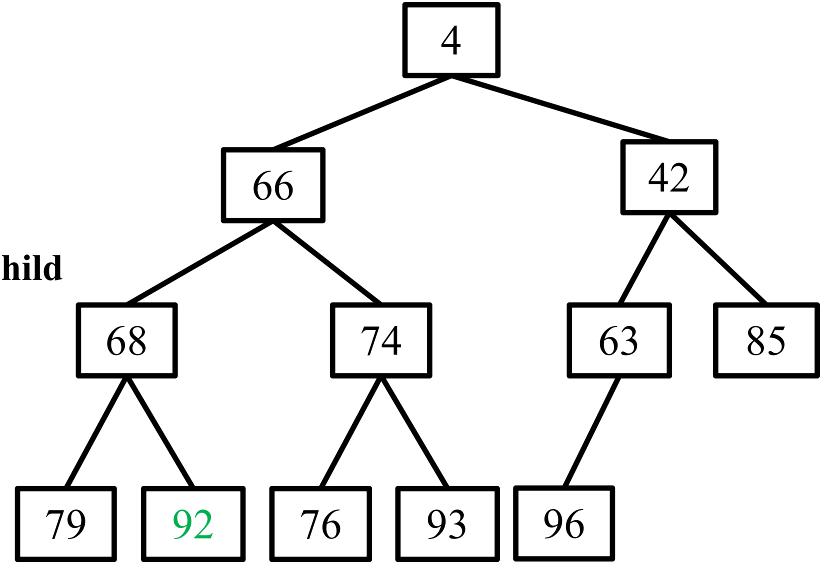
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - **if child is prior, swap with prior child**
  - *siftup*
    - if prior to parent, swap with parent



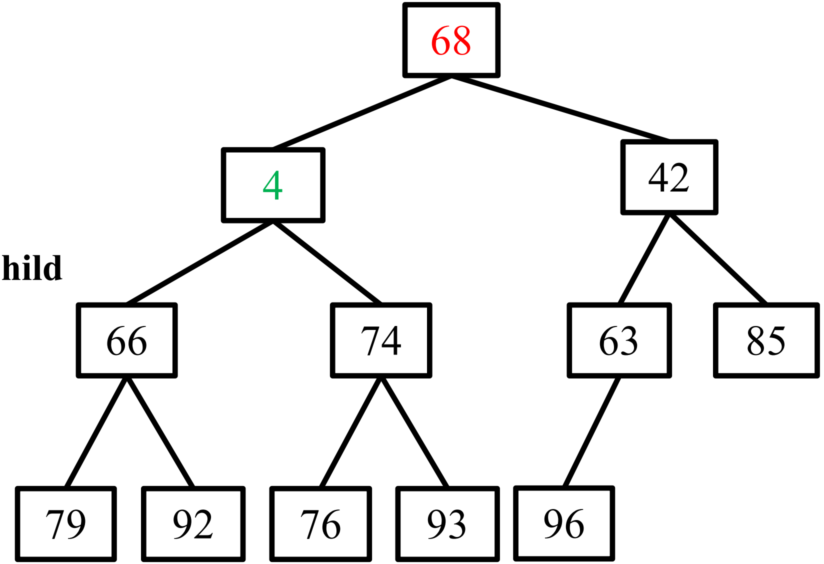
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - **if child is prior, swap with prior child**
  - *siftup*
    - if prior to parent, swap with parent



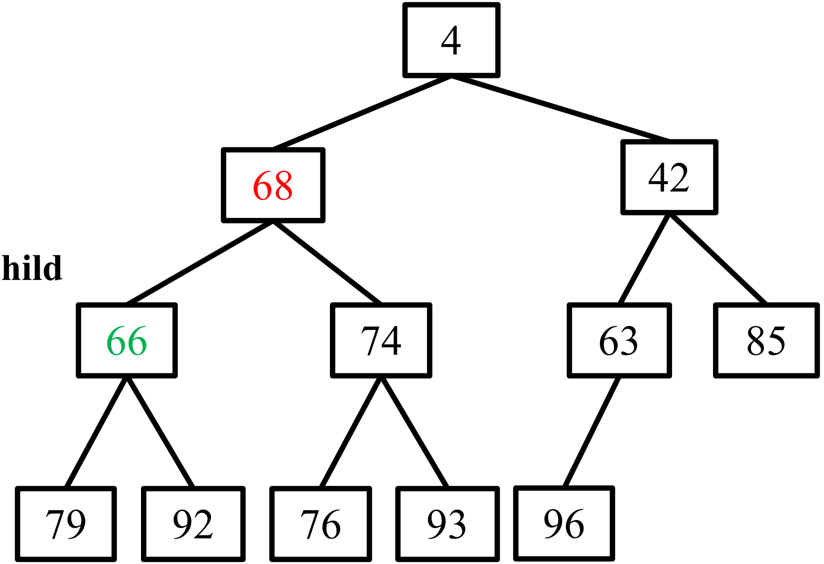
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - **if child is prior, swap with prior child**
  - *siftup*
    - if prior to parent, swap with parent



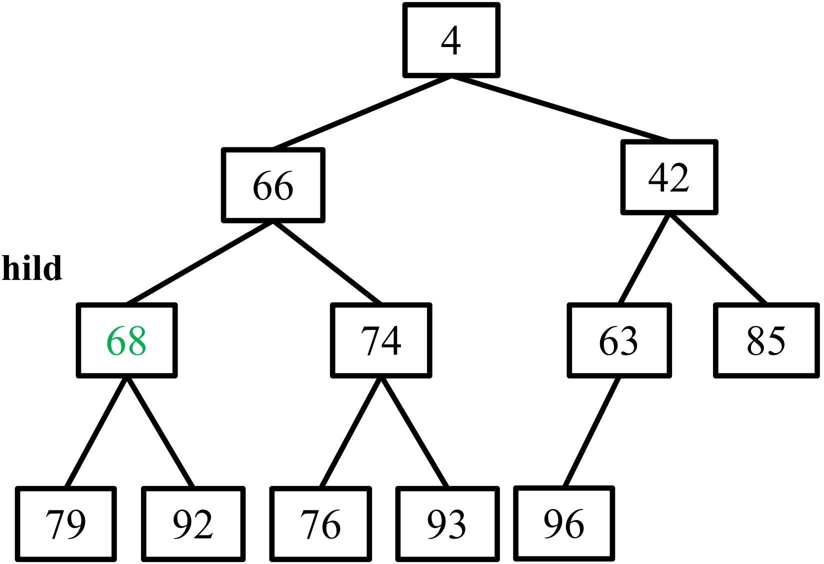
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - **if child is prior, swap with prior child**
  - *siftup*
    - if prior to parent, swap with parent



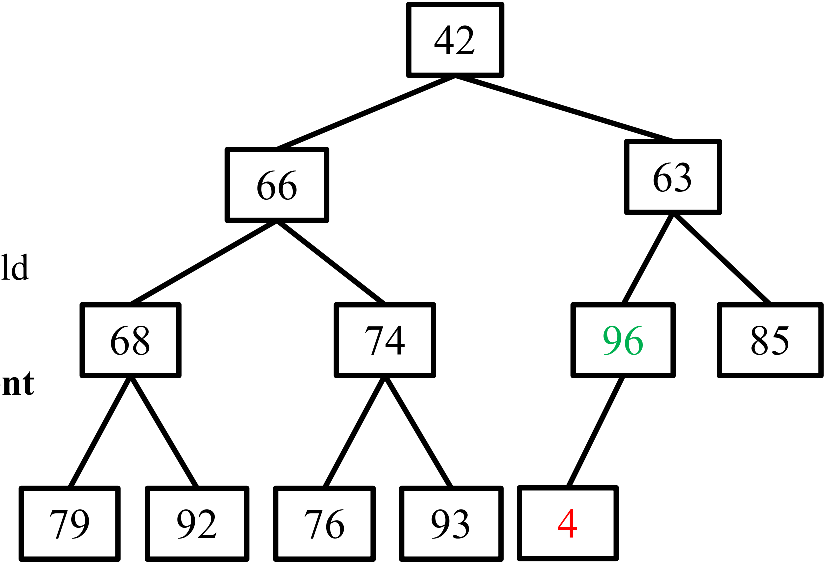
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - **if child is prior, swap with prior child**
  - *siftup*
    - if prior to parent, swap with parent



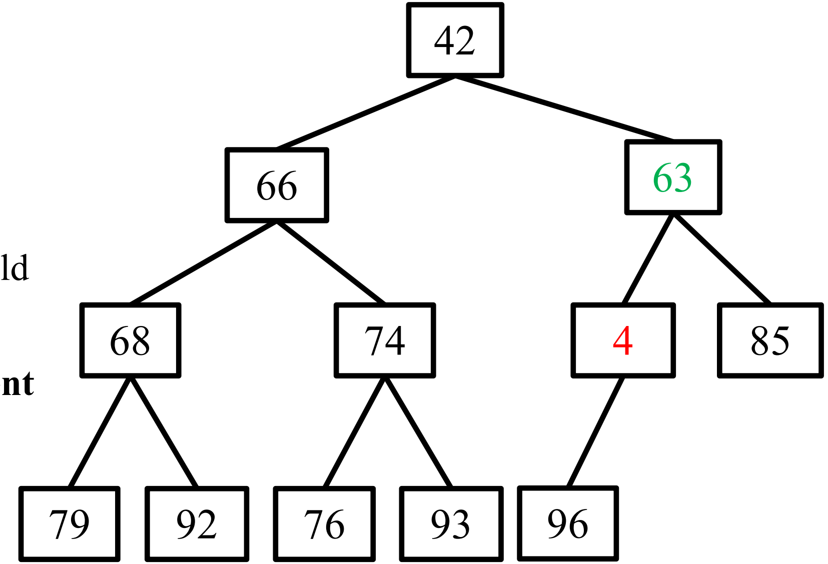
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - **if prior to parent, swap with parent**



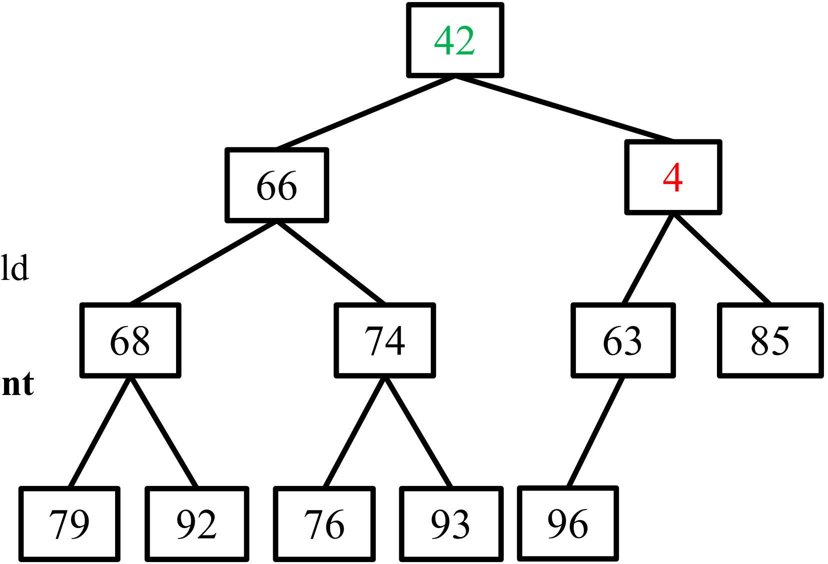
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - **if prior to parent, swap with parent**



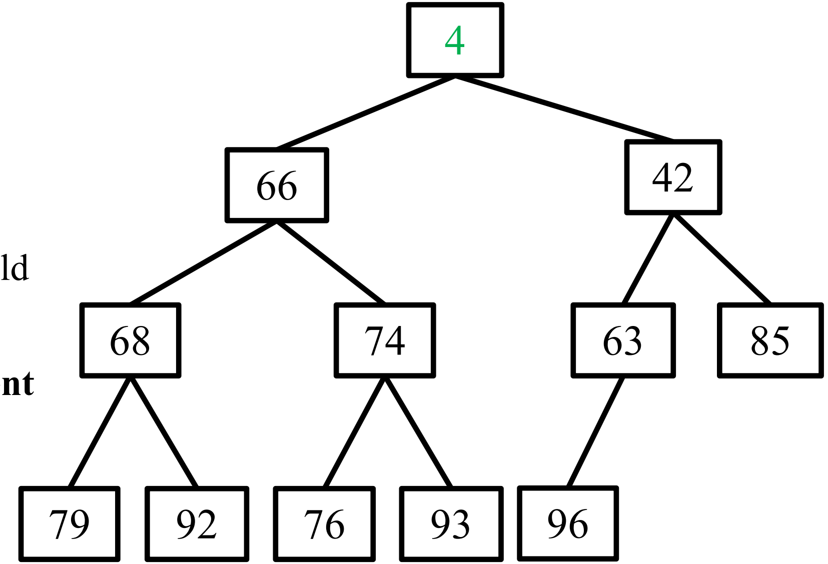
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - **if prior to parent, swap with parent**



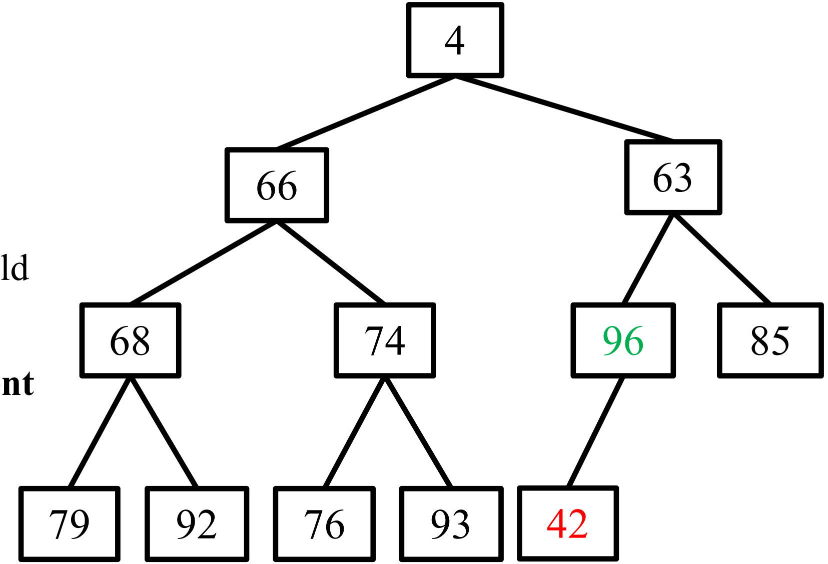
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - **if prior to parent, swap with parent**



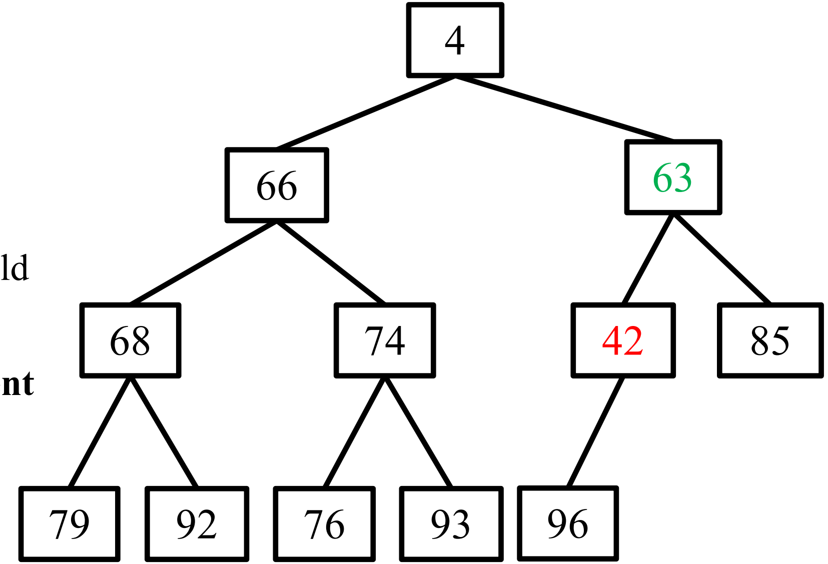
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - **if prior to parent, swap with parent**



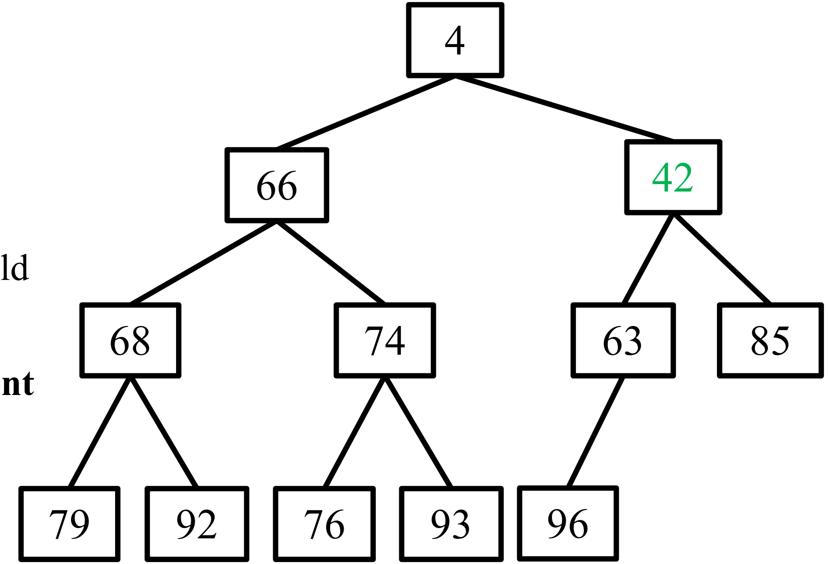
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - **if prior to parent, swap with parent**



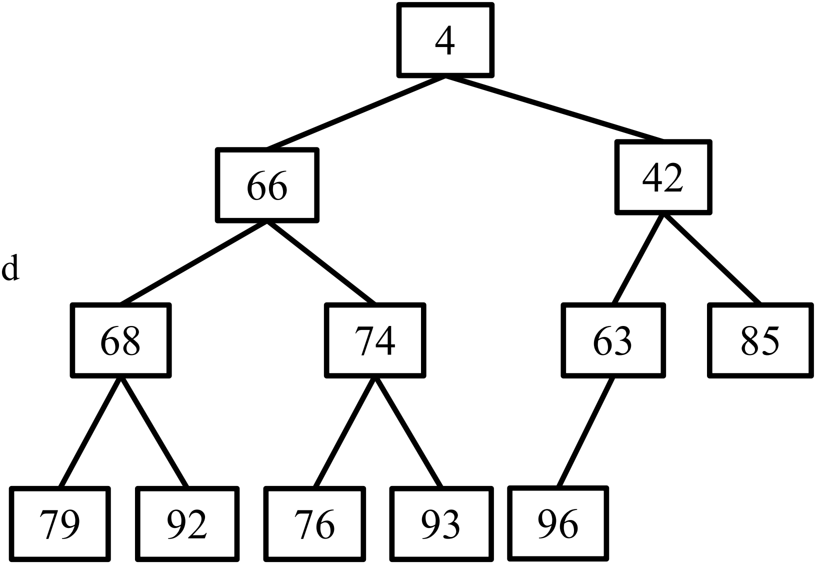
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - **if prior to parent, swap with parent**



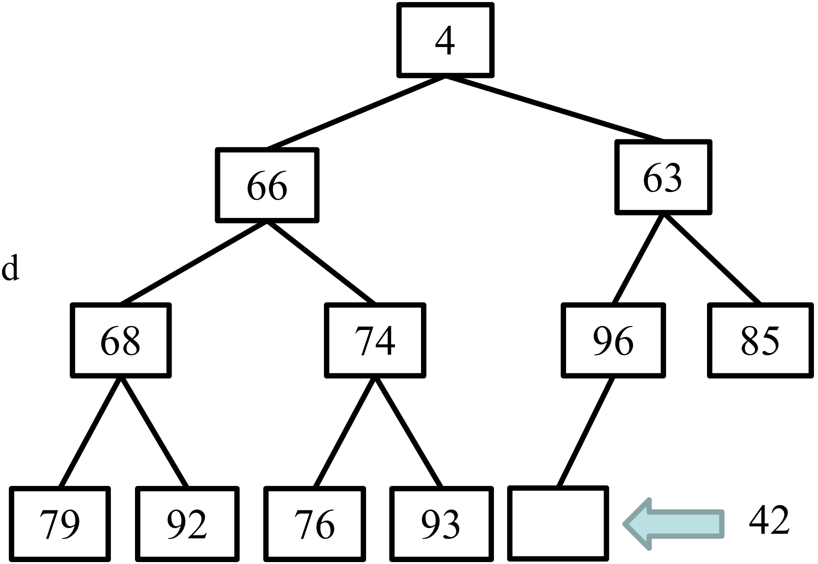
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown



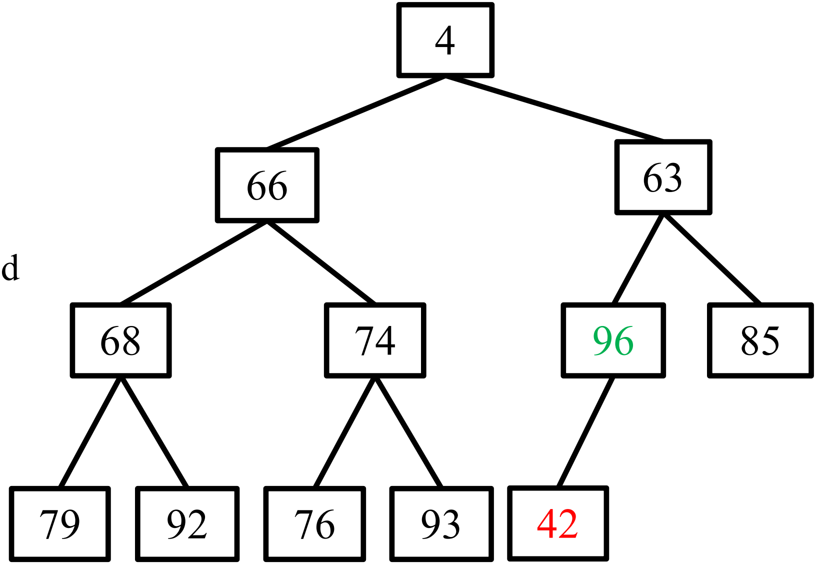
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - **append to end & siftup**
  - *remove* - swap with end & siftdown



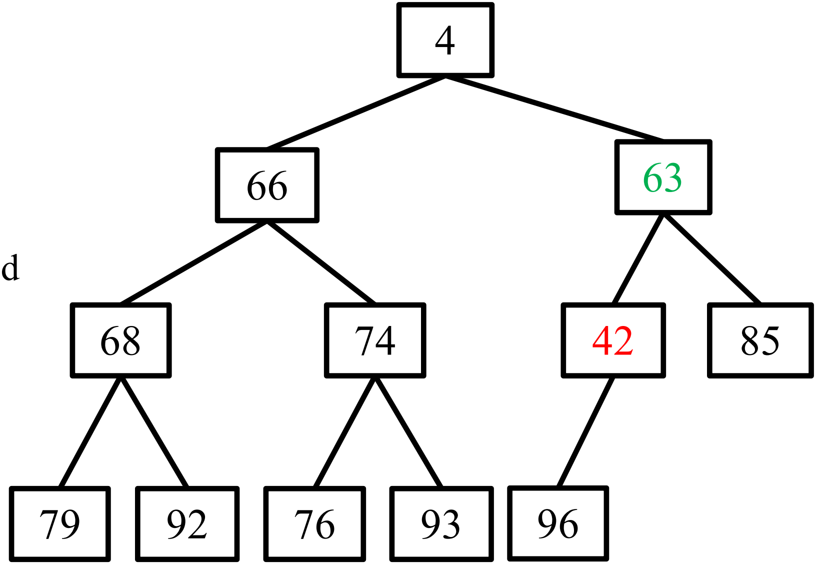
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - **append to end & siftup**
  - *remove* - swap with end & siftdown



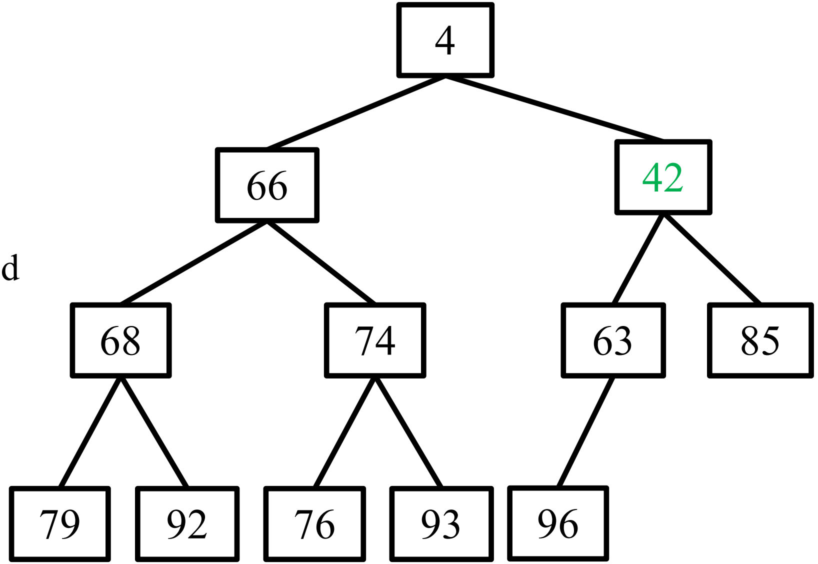
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - **append to end & siftup**
  - *remove* - swap with end & siftdown



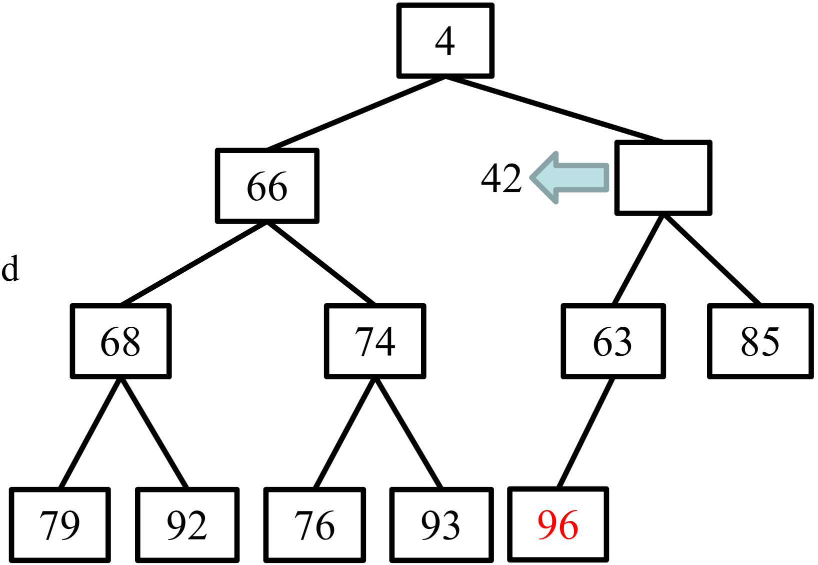
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - **append to end & siftup**
  - *remove* - swap with end & siftdown



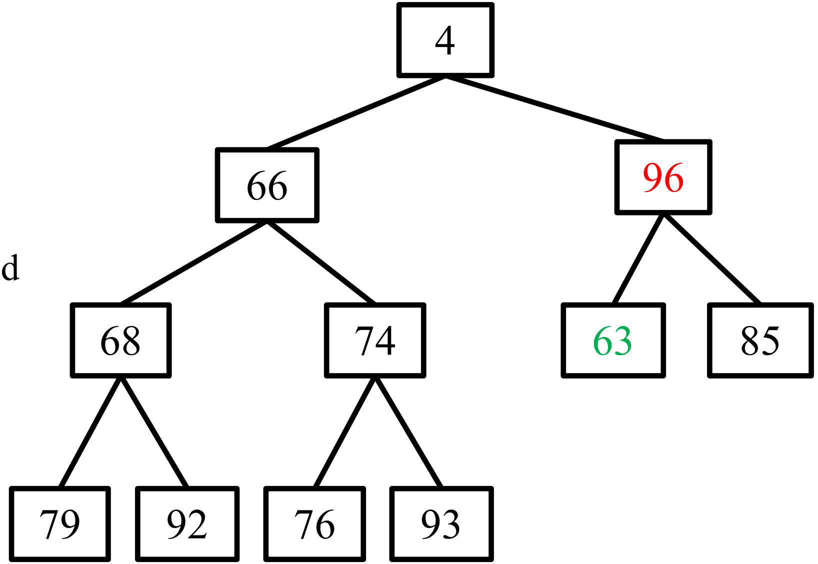
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - append to end & siftup
  - *remove* - **swap with end & siftdown**



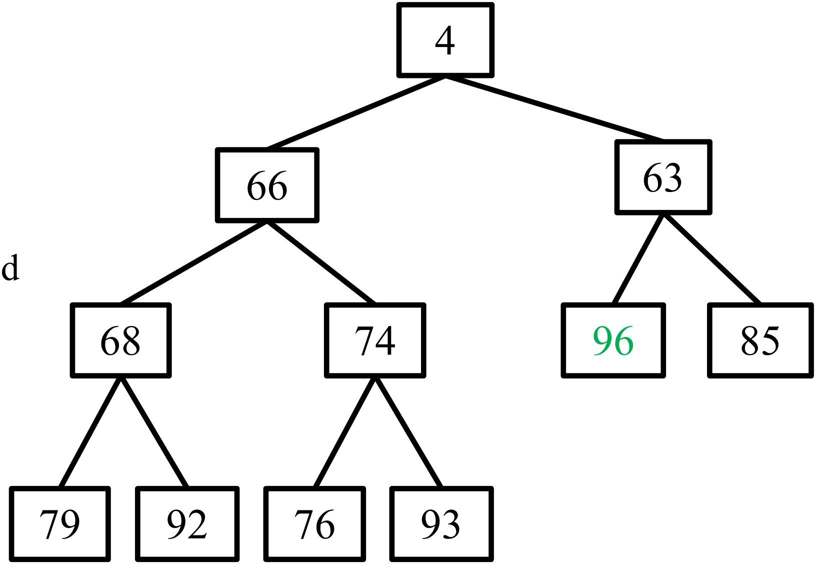
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - append to end & siftup
  - *remove* - **swap with end & siftdown**



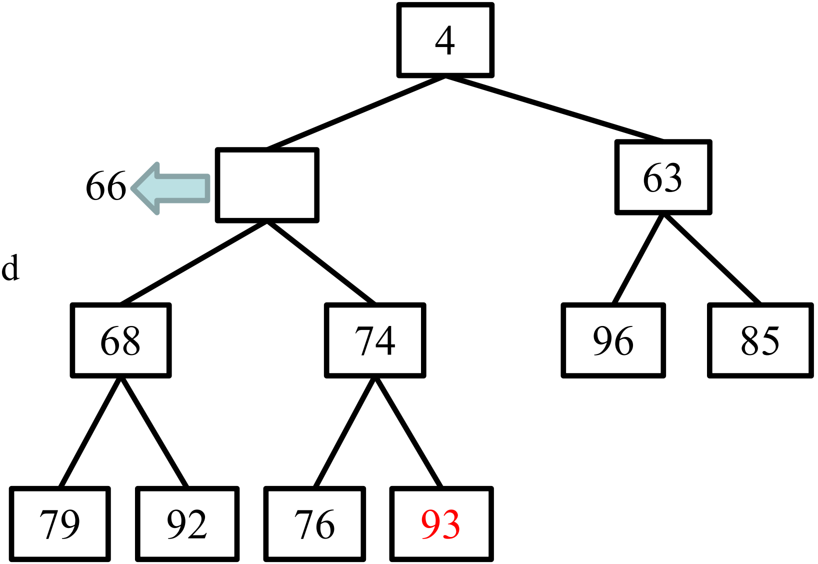
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - append to end & siftup
  - *remove* - **swap with end & siftdown**



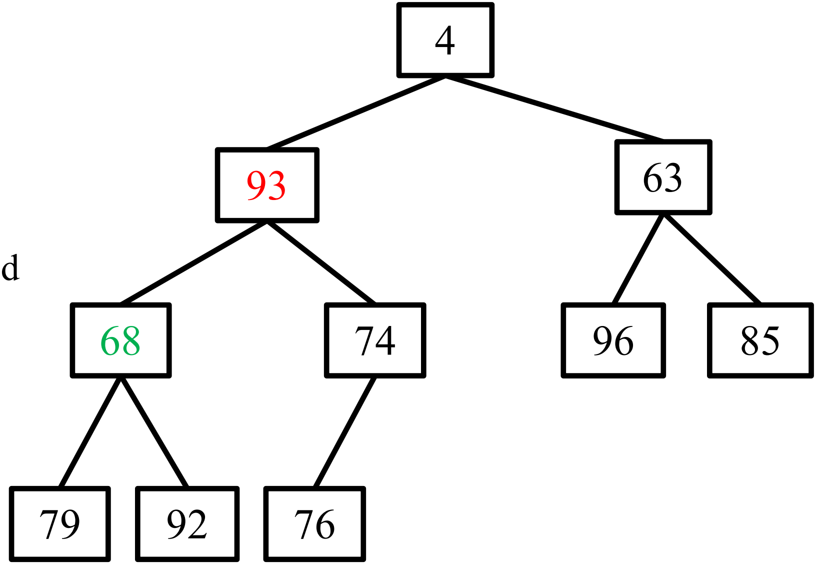
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - append to end & siftup
  - *remove* - **swap with end & siftdown**



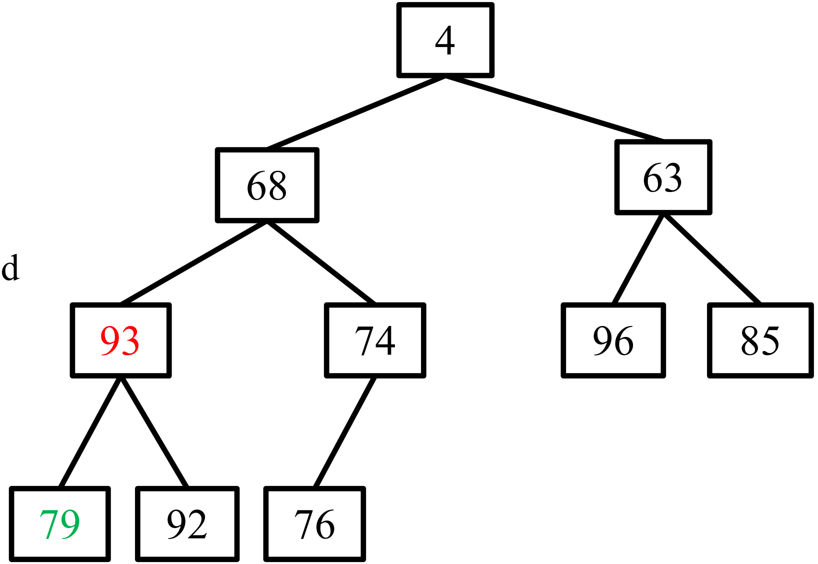
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - append to end & siftup
  - *remove* - **swap with end & siftdown**



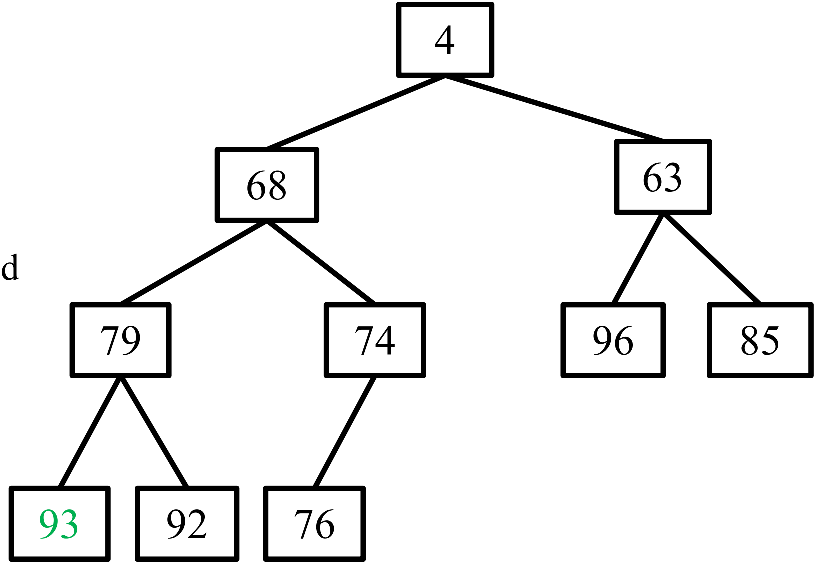
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - append to end & siftup
  - *remove* - **swap with end & siftdown**



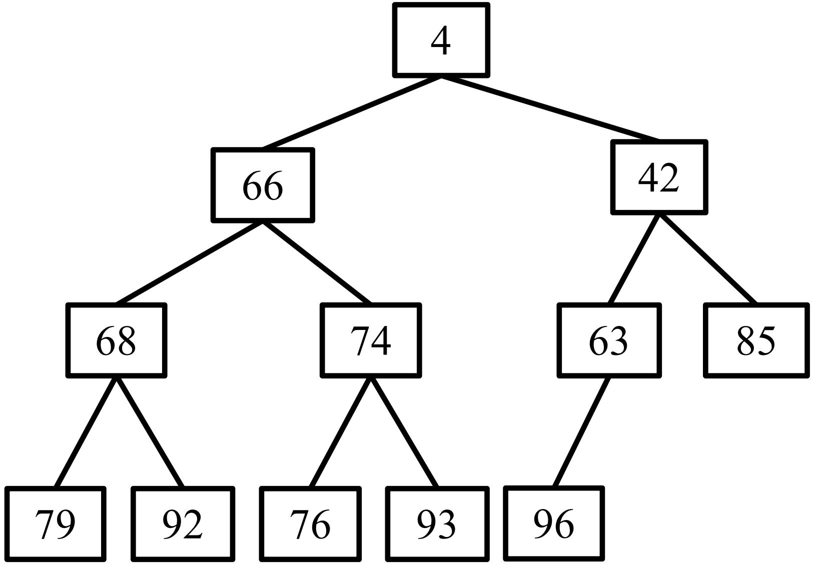
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
    - if child is prior, swap with prior child
  - *siftup*
    - if prior to parent, swap with parent
  - *insert* - append to end & siftup
  - *remove* - **swap with end & siftdown**



Heap

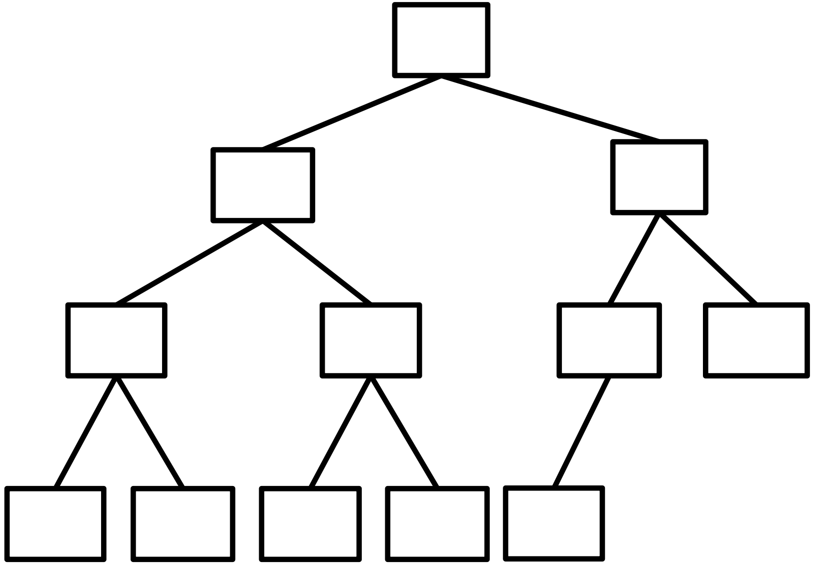
- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - no need to insert one by one



Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - **batch insert**
    - backward iterated siftdown

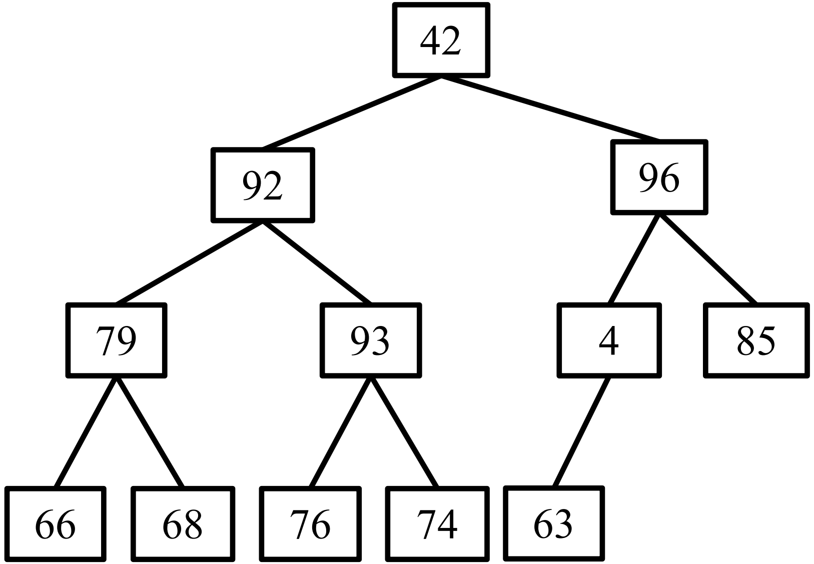
42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



Heap

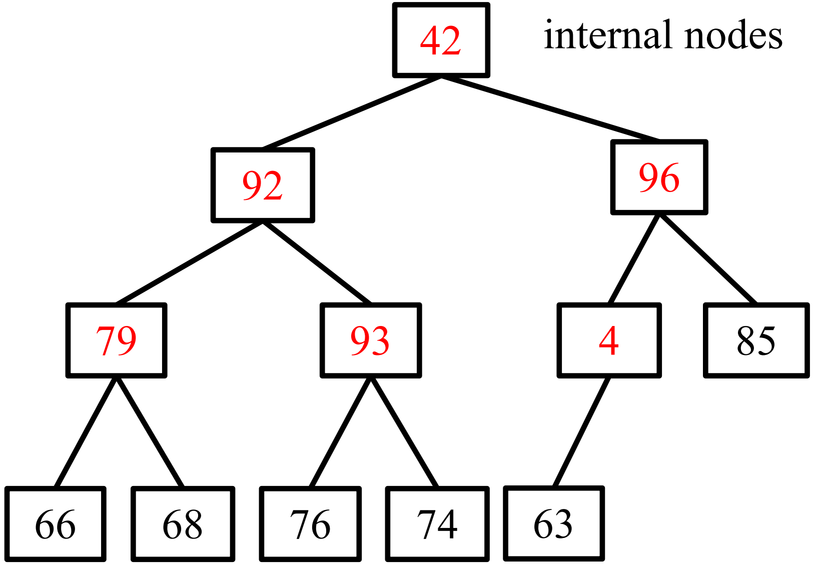
- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - **batch insert**
    - backward iterated siftdown

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



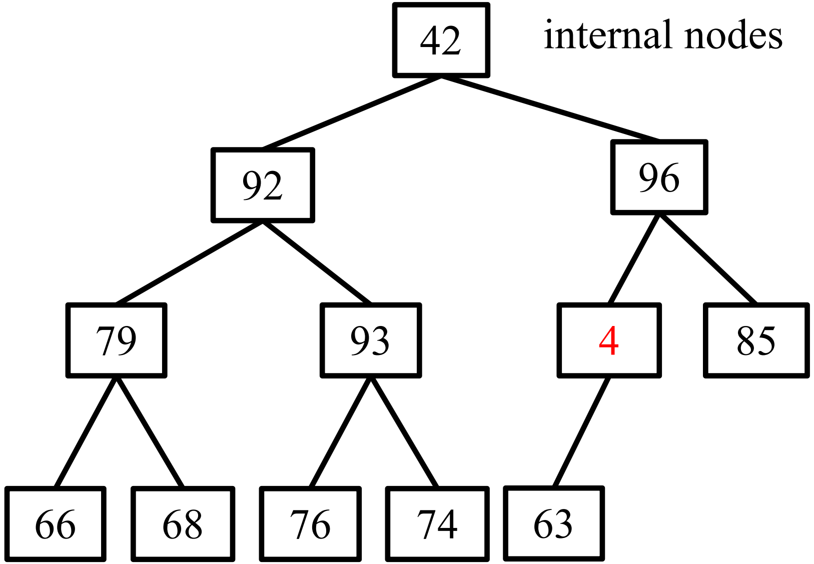
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



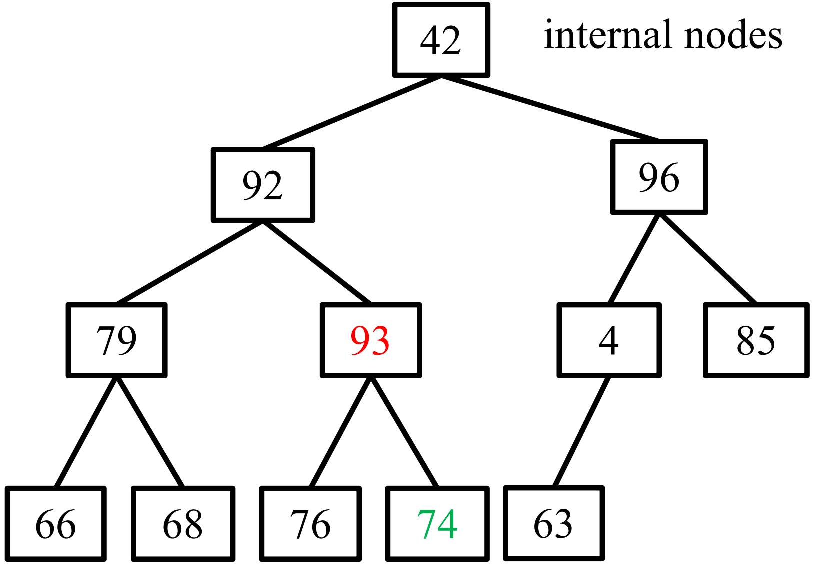
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



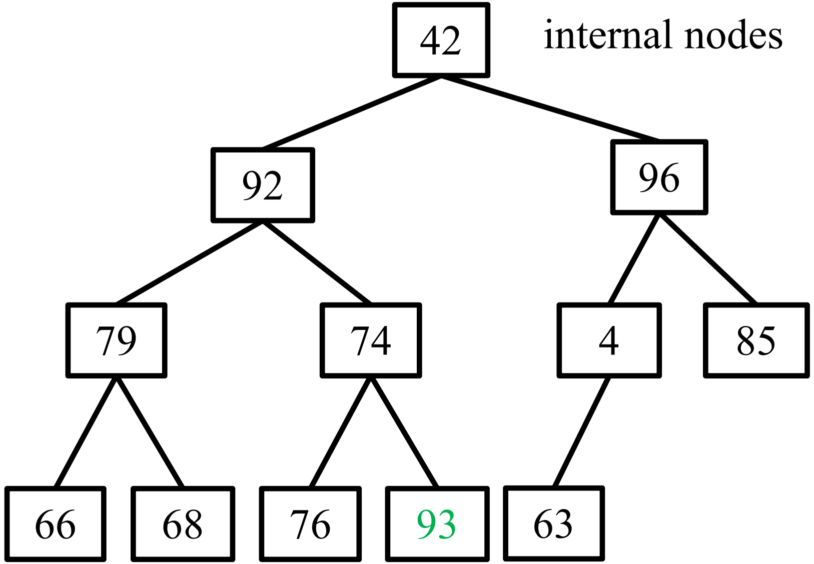
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



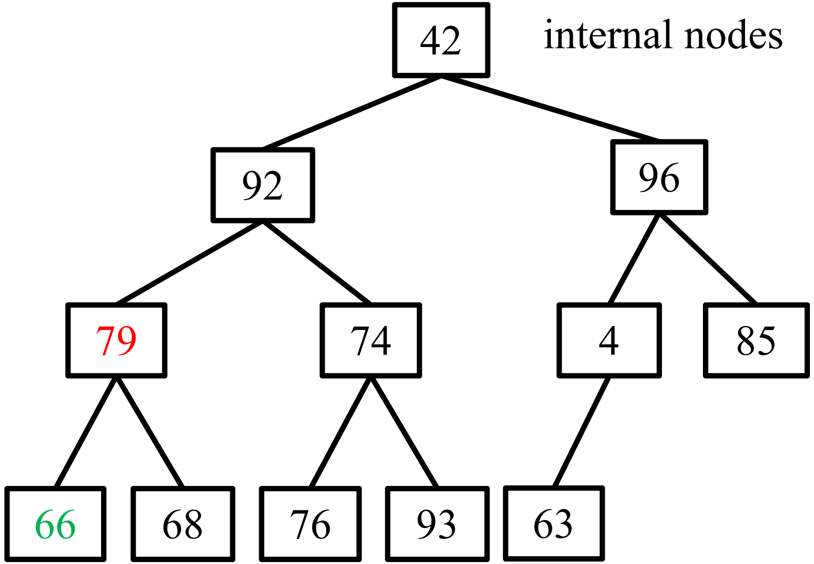
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



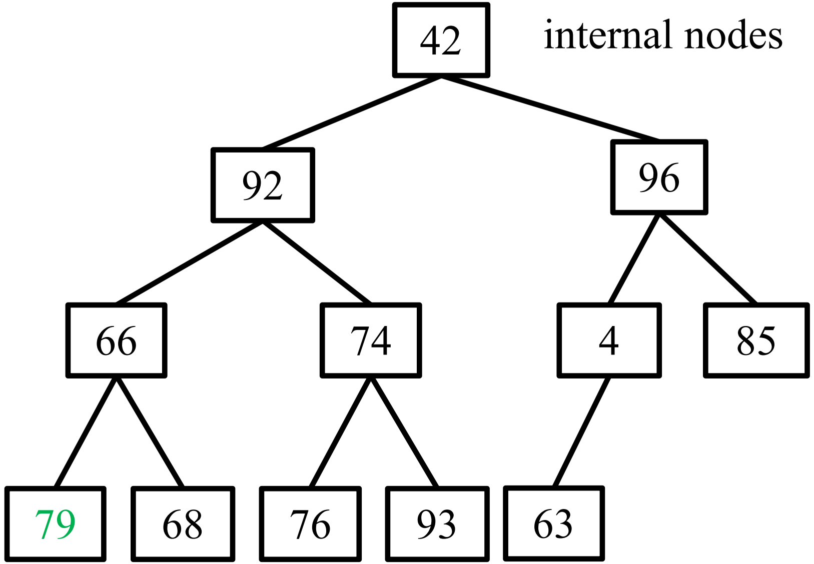
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



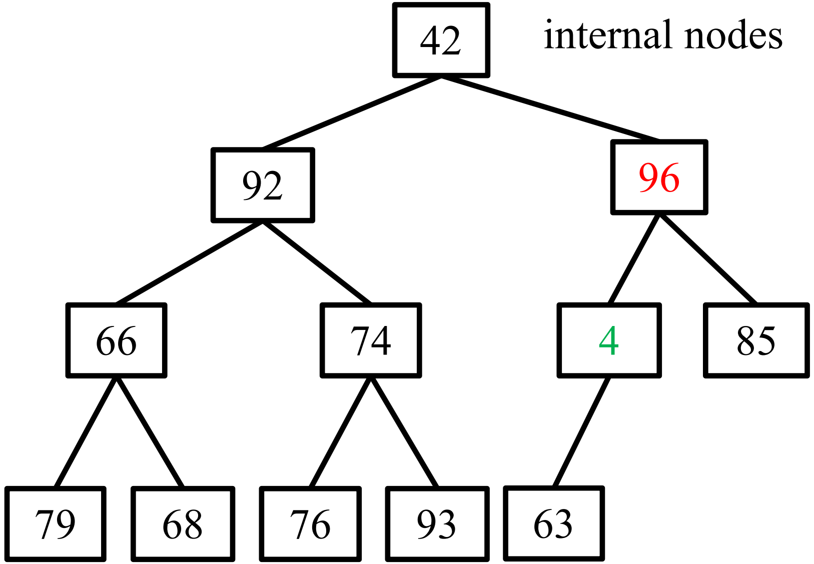
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



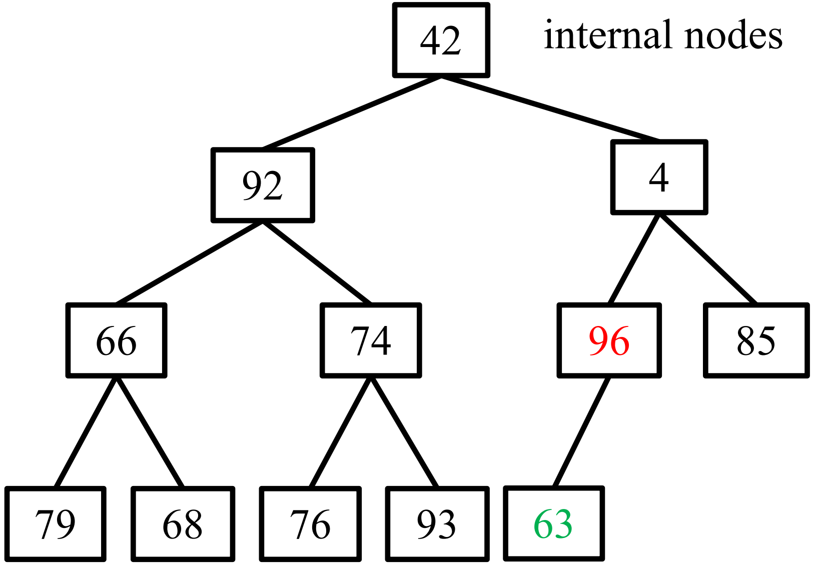
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



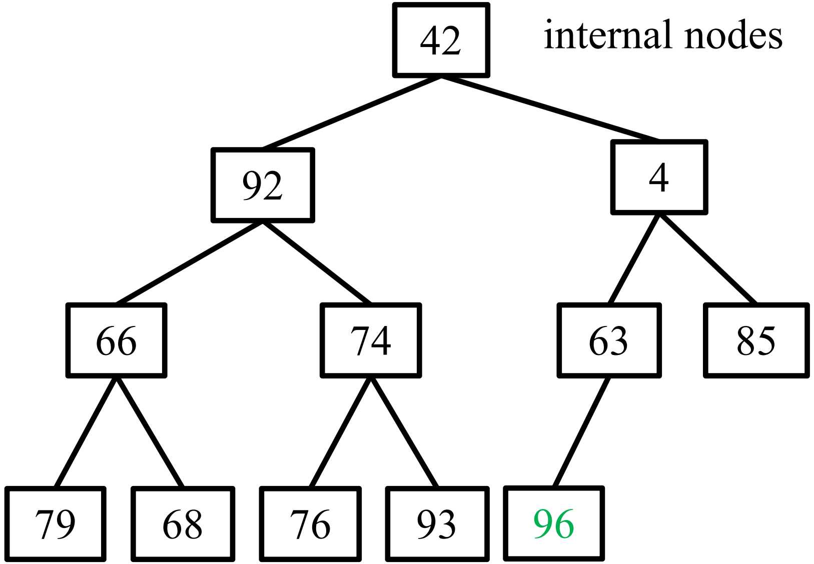
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



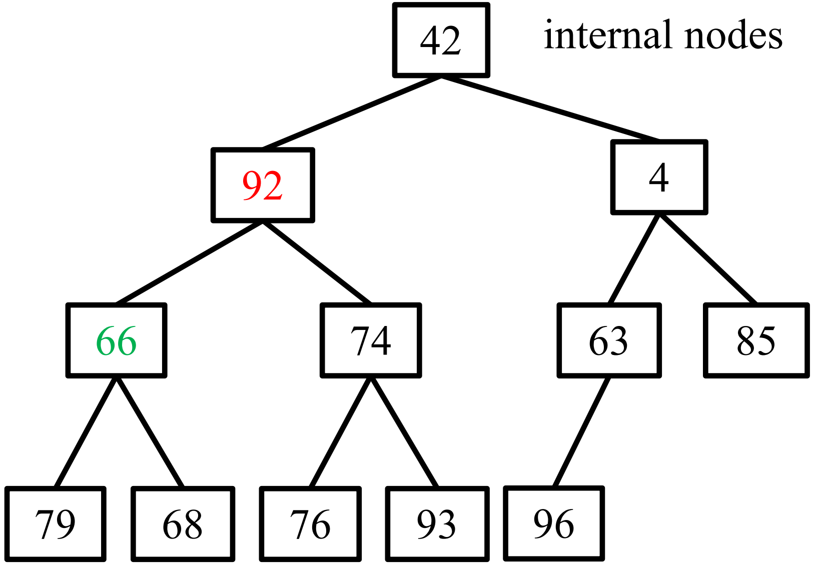
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



Heap

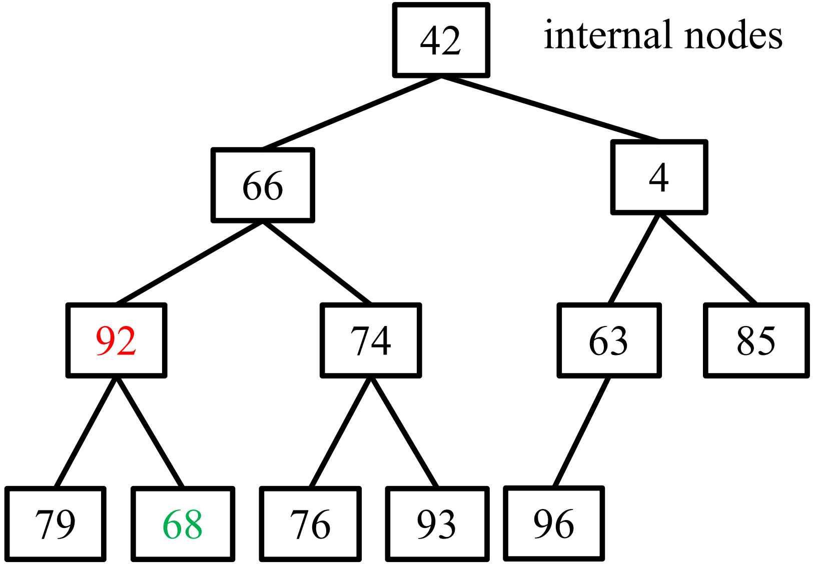
- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



internal nodes

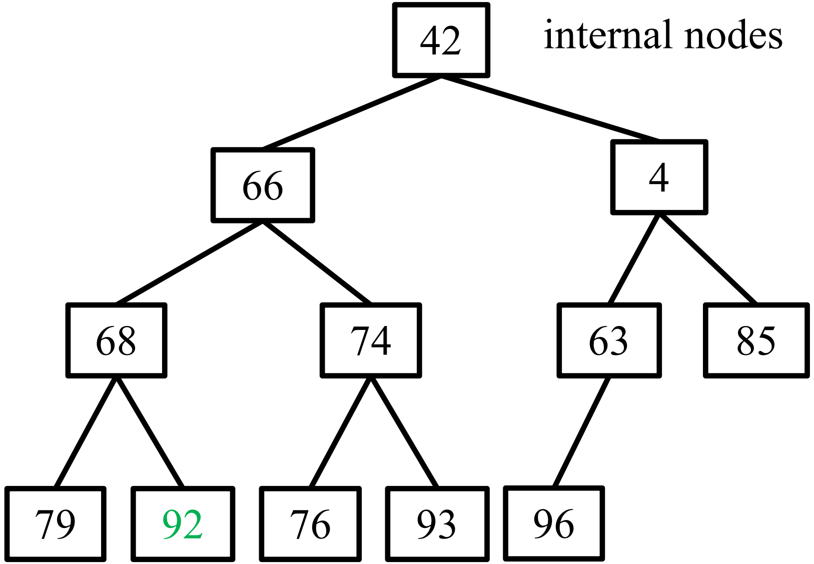
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



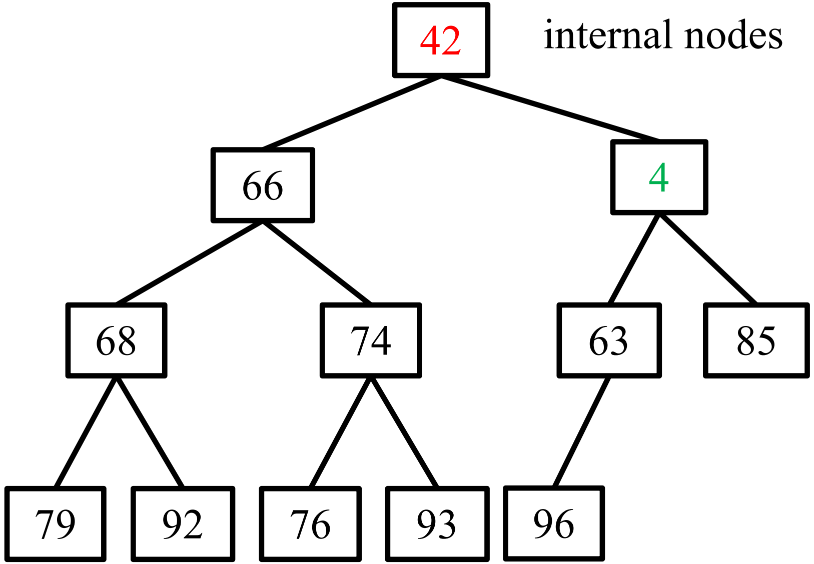
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



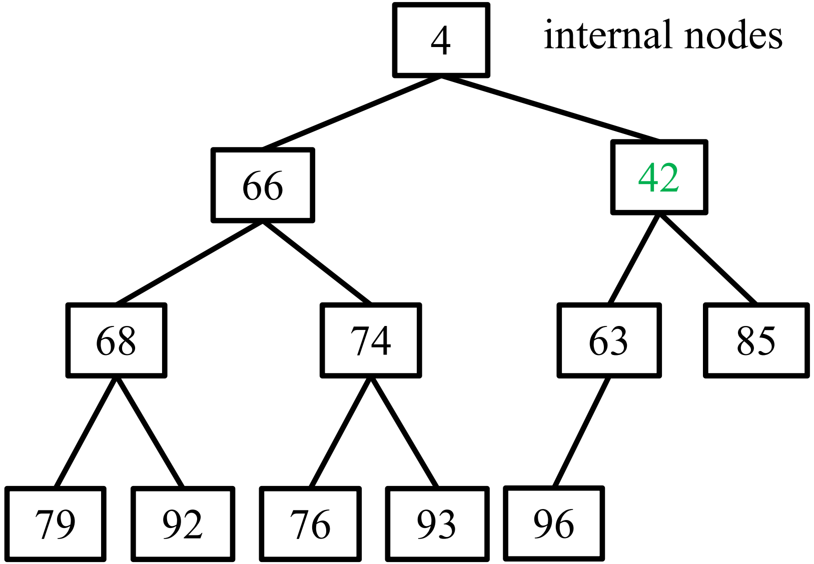
Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



Heap

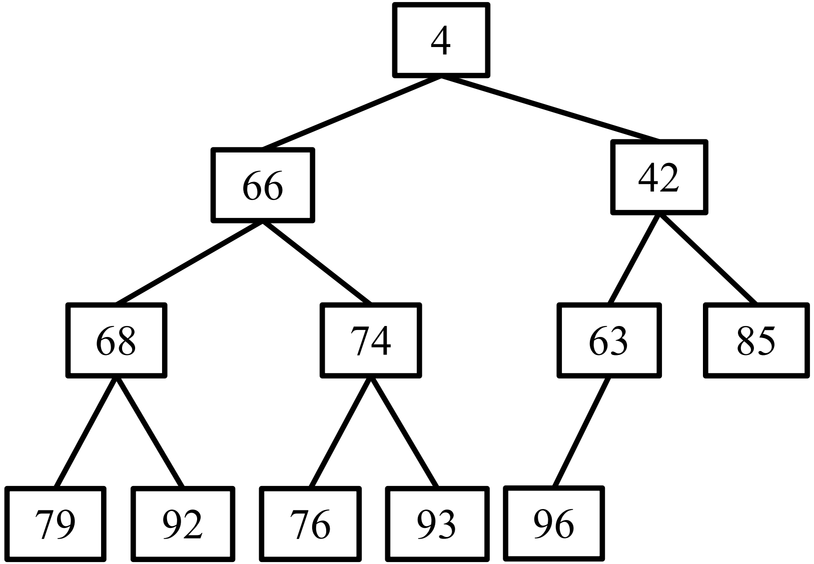
- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*
    - batch insert
    - **backward iterated siftdown**



Heap

- **Heap property - partial order**
  - parent<children : min-heap
  - parent>children : max-heap
- **Complete binary tree**
  - array-based implementation
- **Key operations**
  - *siftdown*
  - *siftup*
  - *insert* - append to end & siftup
  - *remove* - swap with end & siftdown
  - *batch initialization*

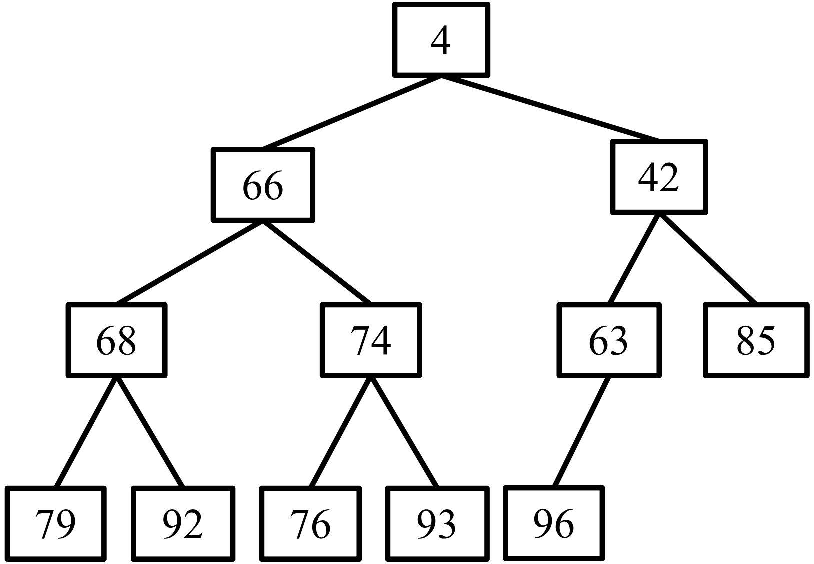
$$c \approx O\left(\sum_{k=1}^{\infty} k \frac{n}{2^{k+1}}\right)=\mathrm{O(n)}$$

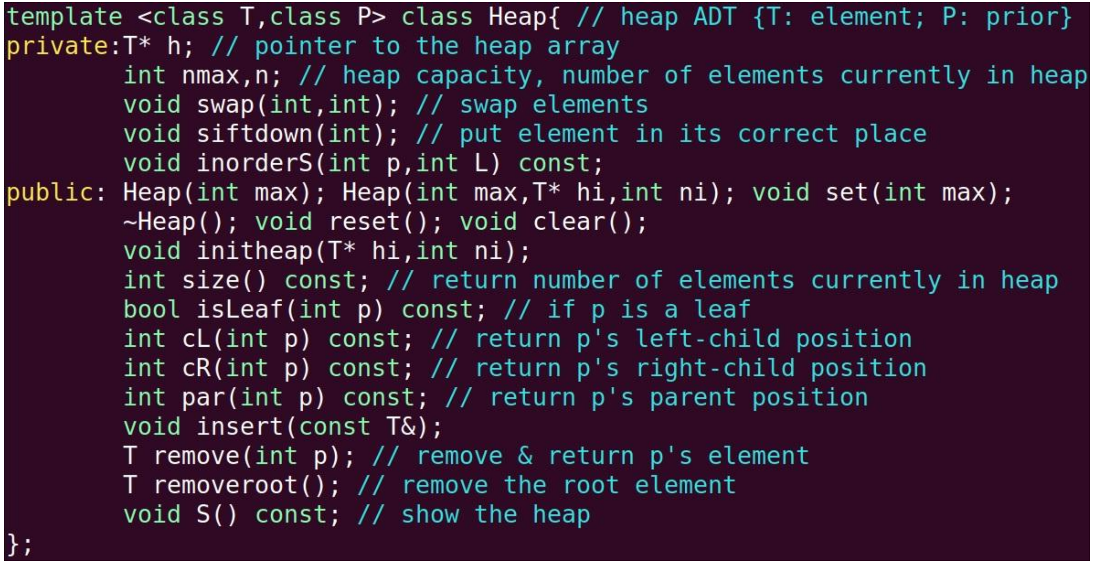


Heap

- **Heap - partial order & complete binary tree (array)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	… …
4	66	42	68	74	63	85	79	92	76	93	96				… …



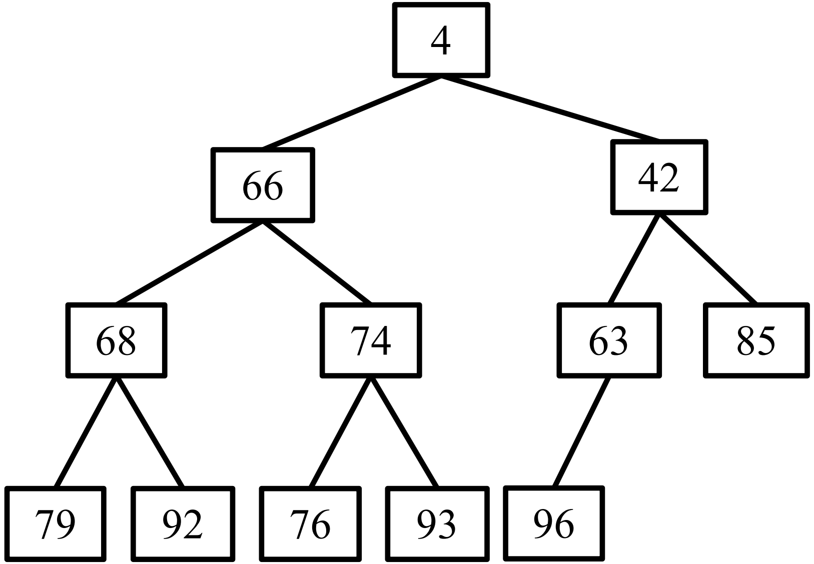


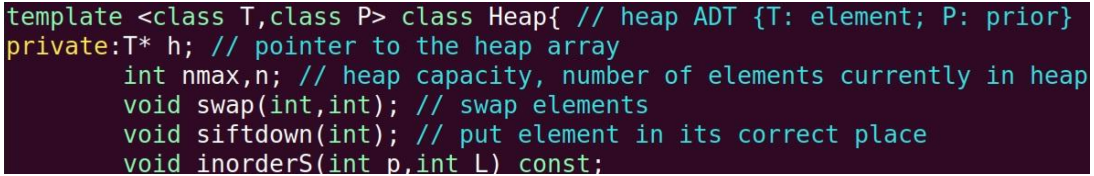
```
template <class T,class P> class Heap{ // heap ADT {T: element; P: prior}
private:T* h; // pointer to the heap array
        int nmax,n; // heap capacity, number of elements currently in heap
        void swap(int,int); // swap elements
        void siftdown(int); // put element in its correct place
        void inorderS(int p,int L) const;
public: Heap(int max); Heap(int max,T* hi,int ni); void set(int max);
        ~Heap(); void reset(); void clear();
        void initheap(T* hi,int ni);
        int size() const; // return number of elements currently in heap
        bool isLeaf(int p) const; // if p is a leaf
        int cL(int p) const; // return p's left-child position
        int cR(int p) const; // return p's right-child position
        int par(int p) const; // return p's parent position
        void insert(const T&);
        T remove(int p); // remove & return p's element
        T removeroot(); // remove the root element
        void S() const; // show the heap
};
```

Heap

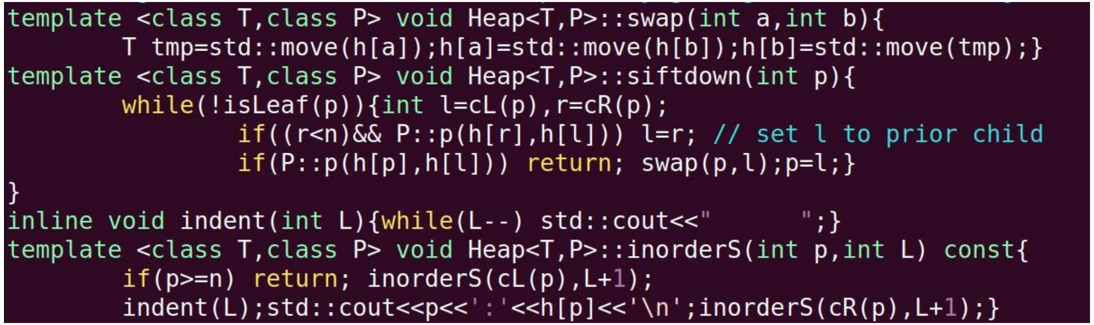
- **Heap - partial order & complete binary tree (array)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	… …
4	66	42	68	74	63	85	79	92	76	93	96				… …





```
template <class T,class P> class Heap{ // heap ADT {T: element; P: prior}
private:T* h; // pointer to the heap array
        int nmax,n; // heap capacity, number of elements currently in heap
        void swap(int,int); // swap elements
        void siftdown(int); // put element in its correct place
        void inorderS(int p,int L) const;
```

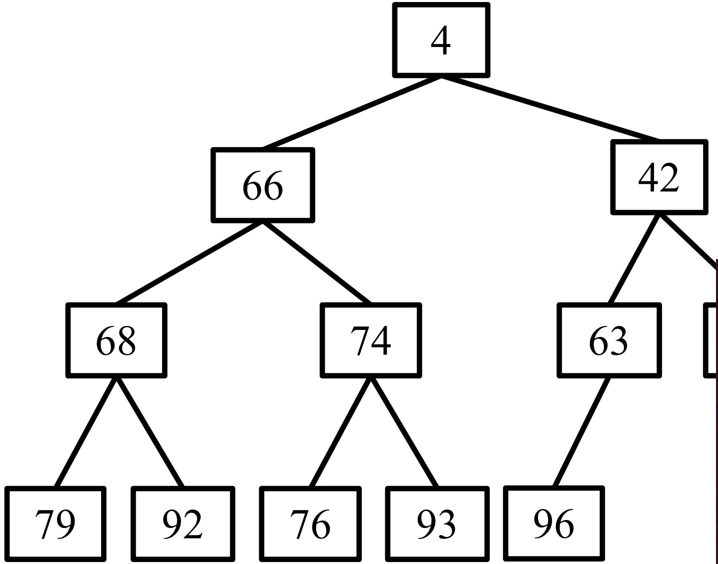


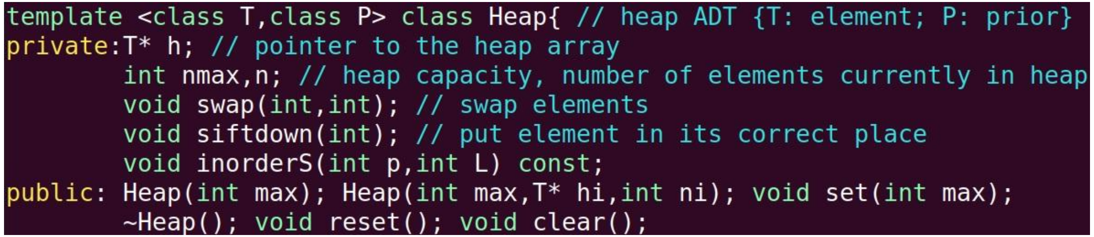
```
template <class T,class P> void Heap<T,P>::swap(int a,int b){
        T tmp=std::move(h[a]);h[a]=std::move(h[b]);h[b]=std::move(tmp);}
template <class T,class P> void Heap<T,P>::siftdown(int p){
        while(!isLeaf(p)){int l=cL(p),r=cR(p);
                if((r<n)&& P::p(h[r],h[l])) l=r; // set l to prior child
                if(P::p(h[p],h[l])) return; swap(p,l);p=l;}
}
inline void indent(int L){while(L--) std::cout<<"      ";}
template <class T,class P> void Heap<T,P>::inorderS(int p,int L) const{
        if(p>=n) return; inorderS(cL(p),L+1);
        indent(L);std::cout<<p<<':'<<h[p]<<'\n';inorderS(cR(p),L+1);}
```

Heap

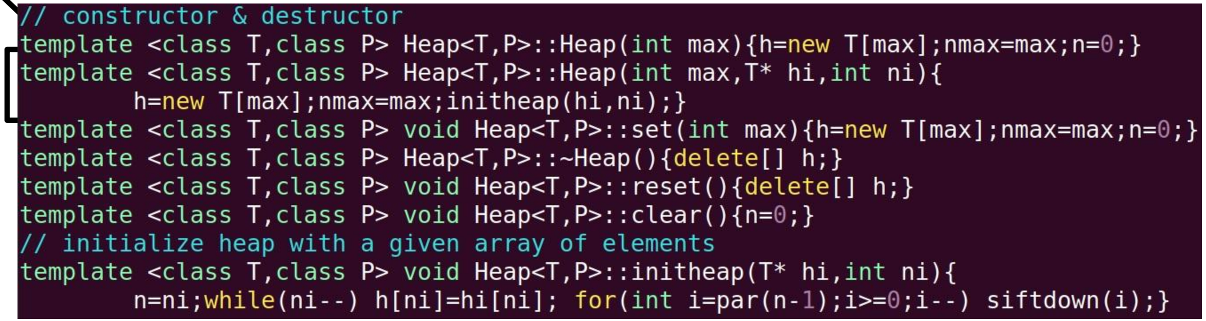
- **Heap - partial order & complete binary tree (array)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	… …
4	66	42	68	74	63	85	79	92	76	93	96				… …





```
template <class T,class P> class Heap{ // heap ADT {T: element; P: prior}
private:T* h; // pointer to the heap array
        int nmax,n; // heap capacity, number of elements currently in heap
        void swap(int,int); // swap elements
        void siftdown(int); // put element in its correct place
        void inorderS(int p,int L) const;
public: Heap(int max); Heap(int max,T* hi,int ni); void set(int max);
        ~Heap(); void reset(); void clear();
```

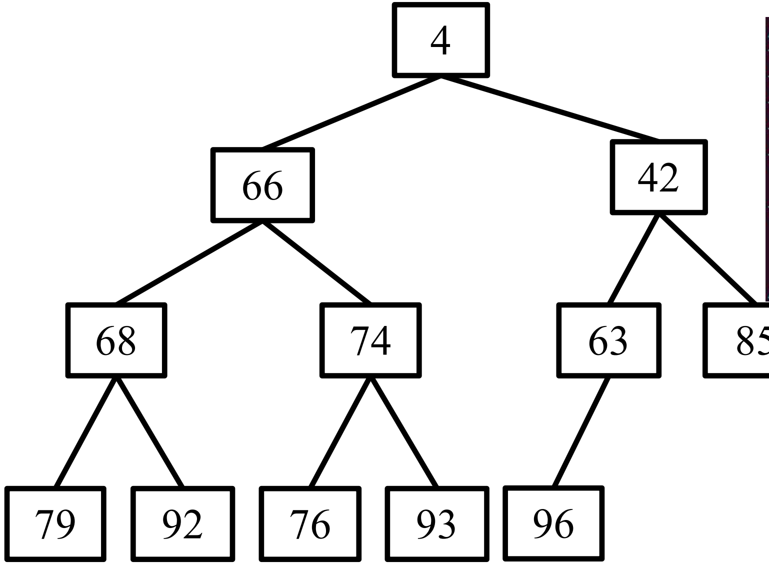


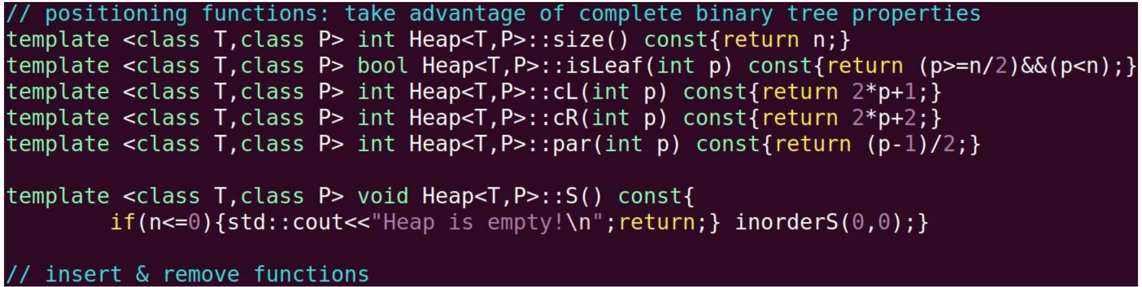
```
// constructor & destructor
template <class T,class P> Heap<T,P>::Heap(int max){h=new T[max];nmax=max;n=0;}
template <class T,class P> Heap<T,P>::Heap(int max,T* hi,int ni){
        h=new T[max];nmax=max;initheap(hi,ni);}
template <class T,class P> void Heap<T,P>::set(int max){h=new T[max];nmax=max;n=0;}
template <class T,class P> Heap<T,P>::~Heap(){delete[] h;}
template <class T,class P> void Heap<T,P>::reset(){delete[] h;}
template <class T,class P> void Heap<T,P>::clear(){n=0;}
// initialize heap with a given array of elements
template <class T,class P> void Heap<T,P>::initheap(T* hi,int ni){
        n=ni;while(ni--) h[ni]=hi[ni]; for(int i=par(n-1);i>=0;i--) siftdown(i);}
```

Heap

- **Heap - partial order & complete binary tree (array)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	… …
4	66	42	68	74	63	85	79	92	76	93	96				… …



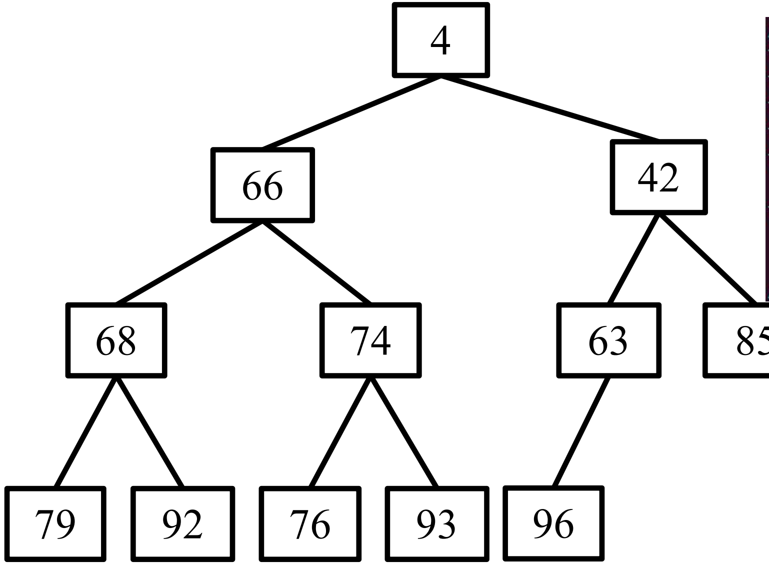


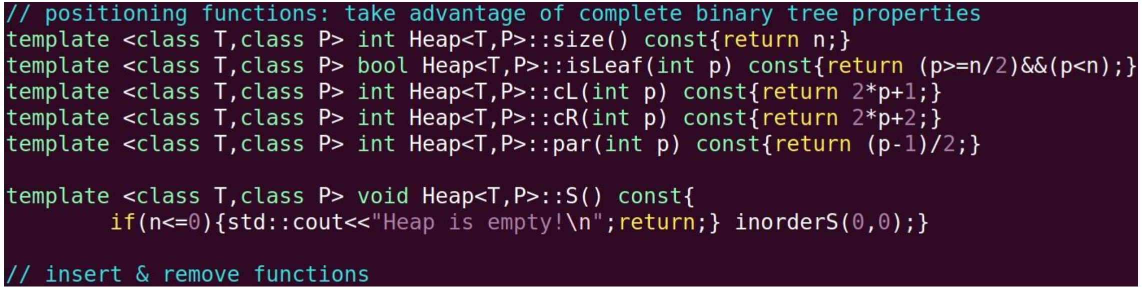
```
// positioning functions: take advantage of complete binary tree properties
template <class T,class P> int Heap<T,P>::size() const{return n;}
template <class T,class P> bool Heap<T,P>::isLeaf(int p) const{return (p>=n/2)&&(p<n);}
template <class T,class P> int Heap<T,P>::cL(int p) const{return 2*p+1;}
template <class T,class P> int Heap<T,P>::cR(int p) const{return 2*p+2;}
template <class T,class P> int Heap<T,P>::par(int p) const{return (p-1)/2;}

template <class T,class P> void Heap<T,P>::S() const{
        if(n<=0){std::cout<<"Heap is empty!\n";return;} inorderS(0,0);}

// insert & remove functions
```

Heap

- **Heap - partial order & complete binary tree (array)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	… …
4	66	42	68	74	63	85	79	92	76	93	96				… …





```
// positioning functions: take advantage of complete binary tree properties
template <class T,class P> int Heap<T,P>::size() const{return n;}
template <class T,class P> bool Heap<T,P>::isLeaf(int p) const{return (p>=n/2)&&(p<n);}
template <class T,class P> int Heap<T,P>::cL(int p) const{return 2*p+1;}
template <class T,class P> int Heap<T,P>::cR(int p) const{return 2*p+2;}
template <class T,class P> int Heap<T,P>::par(int p) const{return (p-1)/2;}

template <class T,class P> void Heap<T,P>::S() const{
        if(n<=0){std::cout<<"Heap is empty!\n";return;} inorderS(0,0);}

// insert & remove functions
```

where are code of ***insert & remove functions*** ?

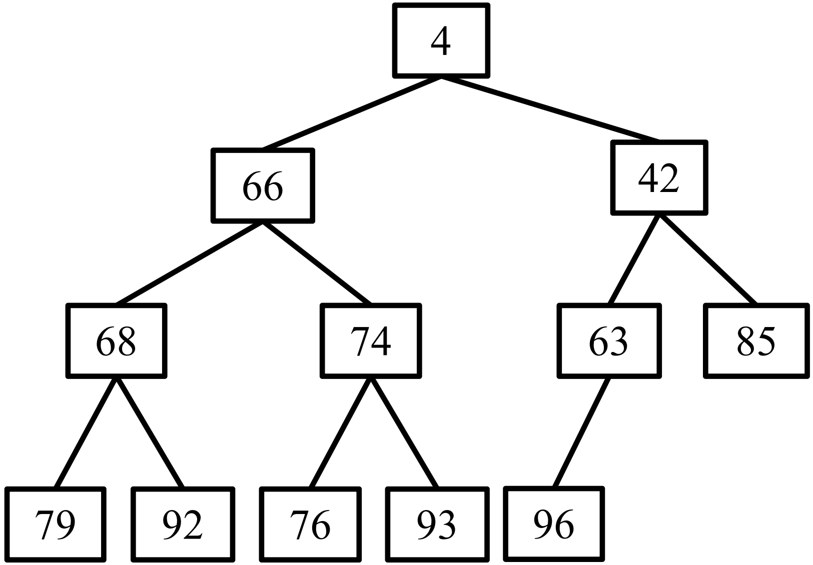
part of homework waiting for you in future

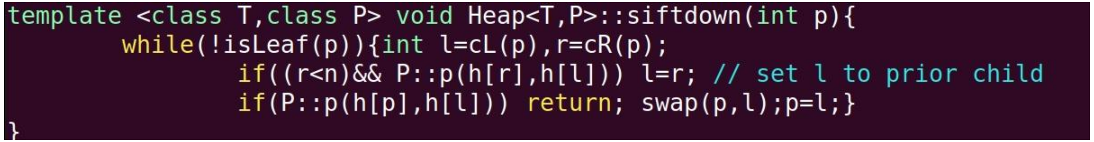
诗与远方，待君自创

Heap

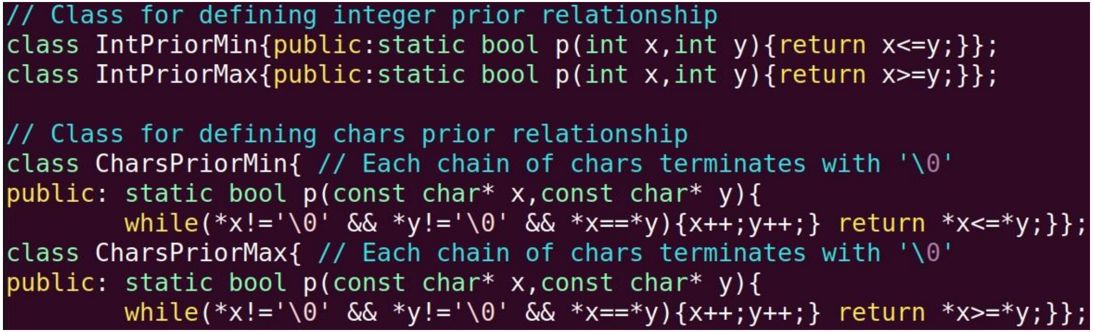
- **Heap - partial order & complete binary tree (array)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	… …
4	66	42	68	74	63	85	79	92	76	93	96				… …





```
template <class T,class P> void Heap<T,P>::siftdown(int p){
        while(!isLeaf(p)){int l=cL(p),r=cR(p);
                if((r<n)&& P::p(h[r],h[l])) l=r; // set l to prior child
                if(P::p(h[p],h[l])) return; swap(p,l);p=l;}
}
```

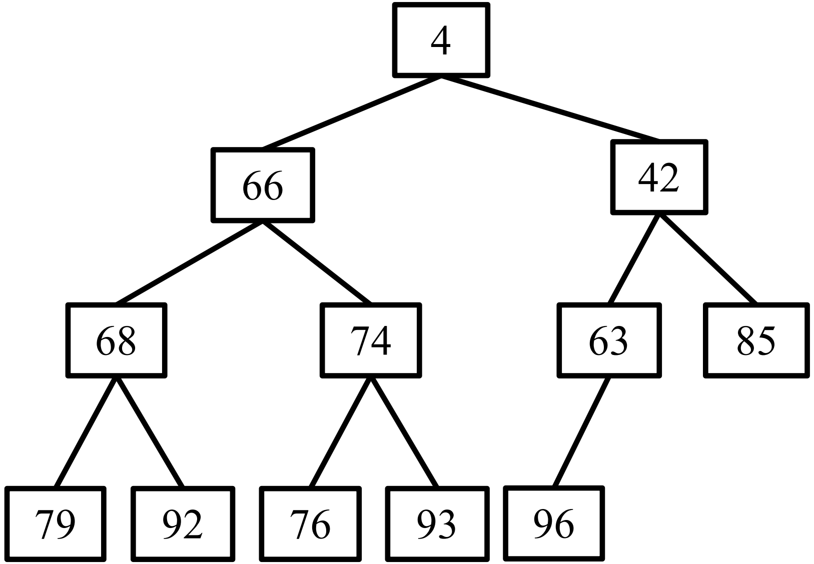


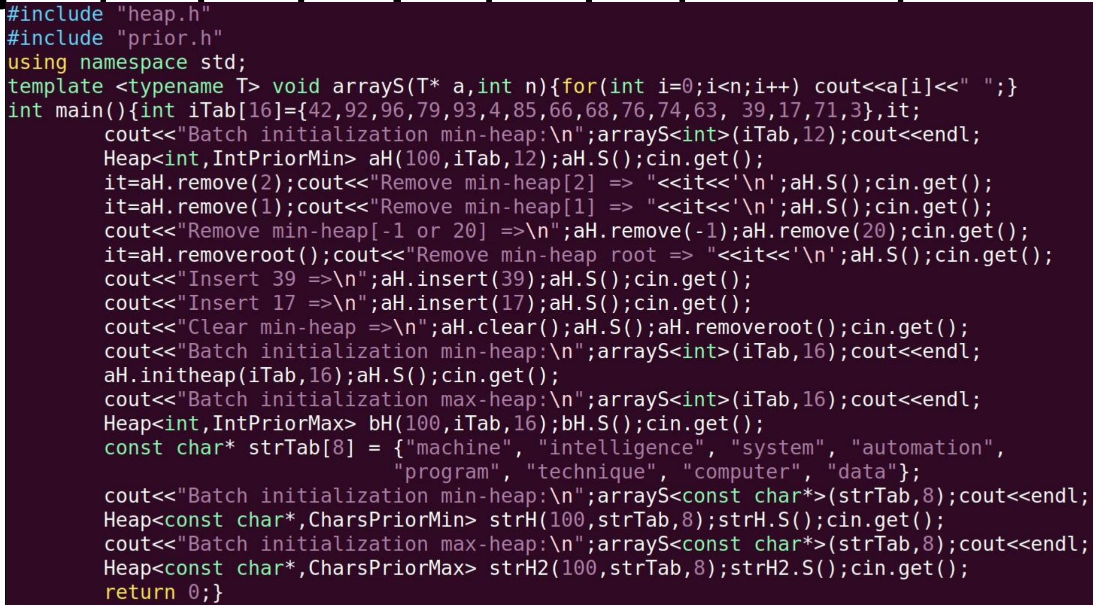
```
// Class for defining integer prior relationship
class IntPriorMin{public:static bool p(int x,int y){return x<=y;}};
class IntPriorMax{public:static bool p(int x,int y){return x>=y;}};

// Class for defining chars prior relationship
class CharsPriorMin{ // Each chain of chars terminates with '\0'
public: static bool p(const char* x,const char* y){
        while(*x!='\0' && *y!='\0' && *x==*y){x++;y++;} return *x<=*y;}};
class CharsPriorMax{ // Each chain of chars terminates with '\0'
public: static bool p(const char* x,const char* y){
        while(*x!='\0' && *y!='\0' && *x==*y){x++;y++;} return *x>=*y;}};
```

Heap

- **Heap - partial order & complete binary tree (array)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	… …
4	66	42	68	74	63	85	79	92	76	93	96				… …

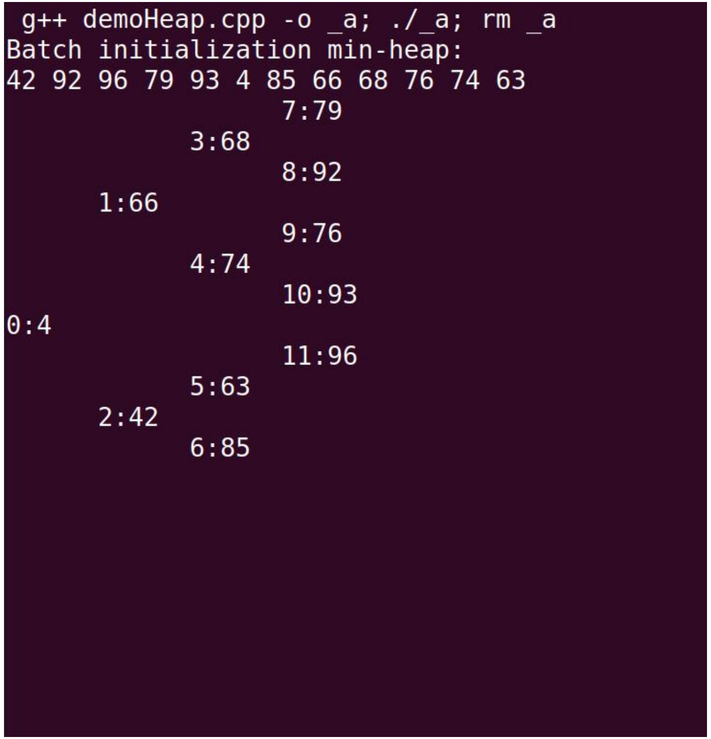




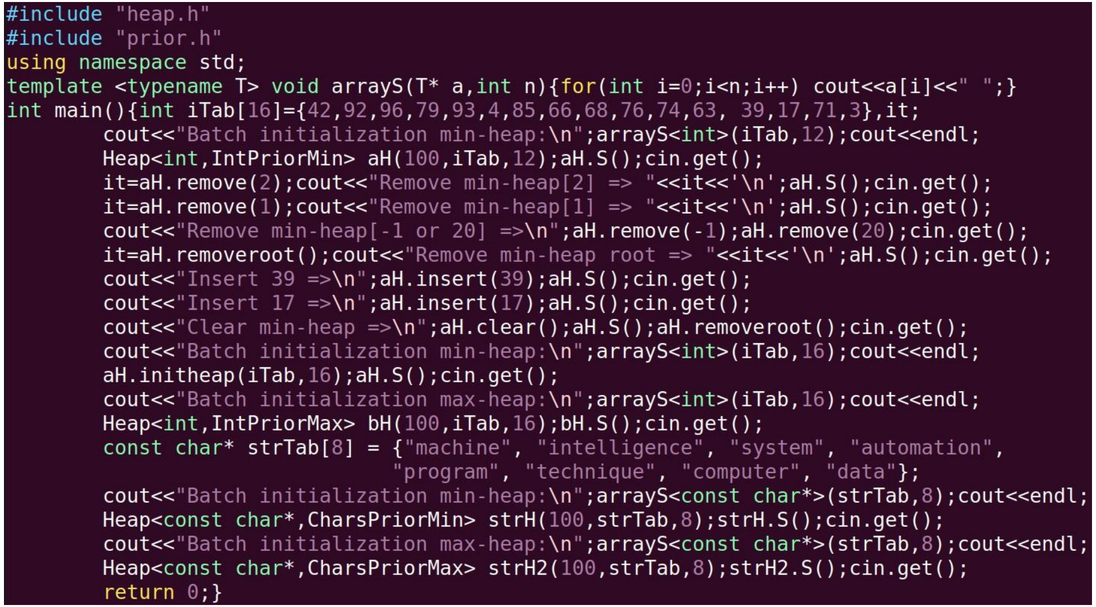
```
#include "heap.h"
#include "prior.h"
using namespace std;
template <typename T> void arrayS(T* a,int n){for(int i=0;i<n;i++) cout<<a[i]<<" ";}
int main(){int iTab[16]={42,92,96,79,93,4,85,66,68,76,74,63, 39,17,71,3},it;
        cout<<"Batch initialization min-heap:\n";arrayS<int>(iTab,12);cout<<endl;
        Heap<int,IntPriorMin> aH(100,iTab,12);aH.S();cin.get();
        it=aH.remove(2);cout<<"Remove min-heap[2] => "<<it<<'\n';aH.S();cin.get();
        it=aH.remove(1);cout<<"Remove min-heap[1] => "<<it<<'\n';aH.S();cin.get();
        cout<<"Remove min-heap[-1 or 20] =>\n";aH.remove(-1);aH.remove(20);cin.get();
        it=aH.removeroot();cout<<"Remove min-heap root => "<<it<<'\n';aH.S();cin.get();
        cout<<"Insert 39 =>\n";aH.insert(39);aH.S();cin.get();
        cout<<"Insert 17 =>\n";aH.insert(17);aH.S();cin.get();
        cout<<"Clear min-heap =>\n";aH.clear();aH.S();aH.removeroot();cin.get();
        cout<<"Batch initialization min-heap:\n";arrayS<int>(iTab,16);cout<<endl;
        aH.initheap(iTab,16);aH.S();cin.get();
        cout<<"Batch initialization max-heap:\n";arrayS<int>(iTab,16);cout<<endl;
        Heap<int,IntPriorMax> bH(100,iTab,16);bH.S();cin.get();
        const char* strTab[8] = {"machine", "intelligence", "system", "automation",
                                "program", "technique", "computer", "data"};
        cout<<"Batch initialization min-heap:\n";arrayS<const char*>(strTab,8);cout<<endl;
        Heap<const char*,CharsPriorMin> strH(100,strTab,8);strH.S();cin.get();
        cout<<"Batch initialization max-heap:\n";arrayS<const char*>(strTab,8);cout<<endl;
        Heap<const char*,CharsPriorMax> strH2(100,strTab,8);strH2.S();cin.get();
        return 0;}
```

Heap

- **Heap - partial order & complete binary tree (array)**



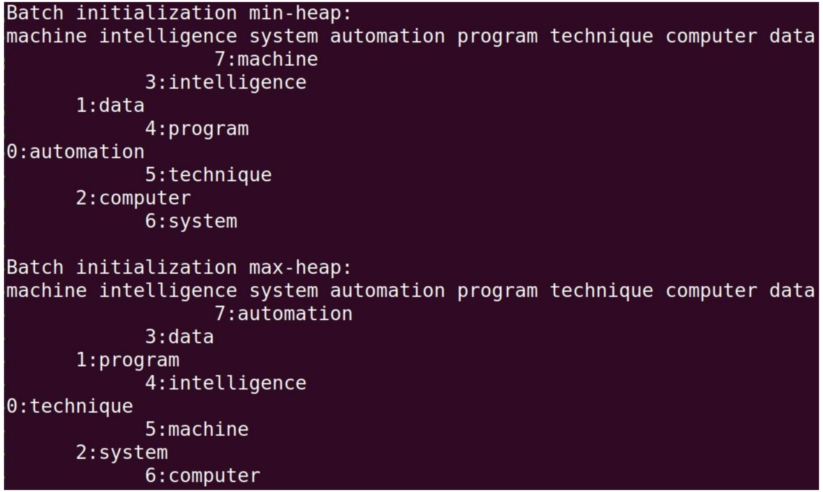
```
 g++ demoHeap.cpp -o _a; ./_a; rm _a
Batch initialization min-heap:
42 92 96 79 93 4 85 66 68 76 74 63
                  7:79
            3:68
                  8:92
      1:66
                  9:76
            4:74
                  10:93
0:4
                  11:96
            5:63
      2:42
            6:85
```



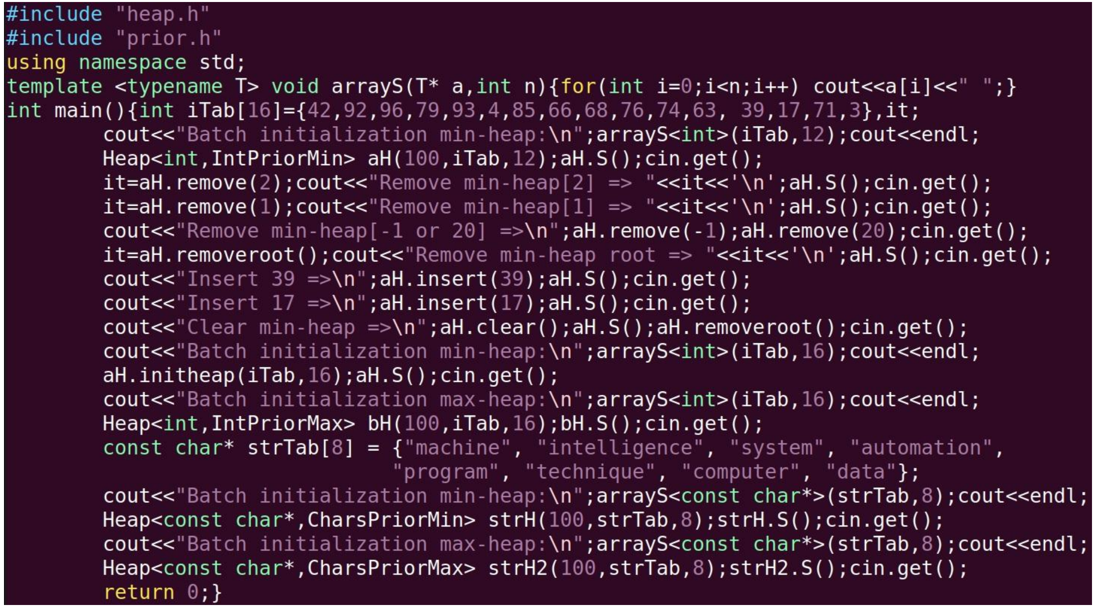
```
#include "heap.h"
#include "prior.h"
using namespace std;
template <typename T> void arrayS(T* a,int n){for(int i=0;i<n;i++) cout<<a[i]<<" ";}
int main(){int iTab[16]={42,92,96,79,93,4,85,66,68,76,74,63, 39,17,71,3},it;
        cout<<"Batch initialization min-heap:\n";arrayS<int>(iTab,12);cout<<endl;
        Heap<int,IntPriorMin> aH(100,iTab,12);aH.S();cin.get();
        it=aH.remove(2);cout<<"Remove min-heap[2] => "<<it<<'\n';aH.S();cin.get();
        it=aH.remove(1);cout<<"Remove min-heap[1] => "<<it<<'\n';aH.S();cin.get();
        cout<<"Remove min-heap[-1 or 20] =>\n";aH.remove(-1);aH.remove(20);cin.get();
        it=aH.removeroot();cout<<"Remove min-heap root => "<<it<<'\n';aH.S();cin.get();
        cout<<"Insert 39 =>\n";aH.insert(39);aH.S();cin.get();
        cout<<"Insert 17 =>\n";aH.insert(17);aH.S();cin.get();
        cout<<"Clear min-heap =>\n";aH.clear();aH.S();aH.removeroot();cin.get();
        cout<<"Batch initialization min-heap:\n";arrayS<int>(iTab,16);cout<<endl;
        aH.initheap(iTab,16);aH.S();cin.get();
        cout<<"Batch initialization max-heap:\n";arrayS<int>(iTab,16);cout<<endl;
        Heap<int,IntPriorMax> bH(100,iTab,16);bH.S();cin.get();
        const char* strTab[8] = {"machine", "intelligence", "system", "automation",
                                "program", "technique", "computer", "data"};
        cout<<"Batch initialization min-heap:\n";arrayS<const char*>(strTab,8);cout<<endl;
        Heap<const char*,CharsPriorMin> strH(100,strTab,8);strH.S();cin.get();
        cout<<"Batch initialization max-heap:\n";arrayS<const char*>(strTab,8);cout<<endl;
        Heap<const char*,CharsPriorMax> strH2(100,strTab,8);strH2.S();cin.get();
        return 0;}
```

Heap

- **Heap - partial order & complete binary tree (array)**



```
Batch initialization min-heap:
machine intelligence system automation program technique computer data
                  7:machine
            3:intelligence
      1:data
            4:program
0:automation
            5:technique
      2:computer
            6:system

Batch initialization max-heap:
machine intelligence system automation program technique computer data
                  7:automation
            3:data
      1:program
            4:intelligence
0:technique
            5:machine
      2:system
            6:computer
```

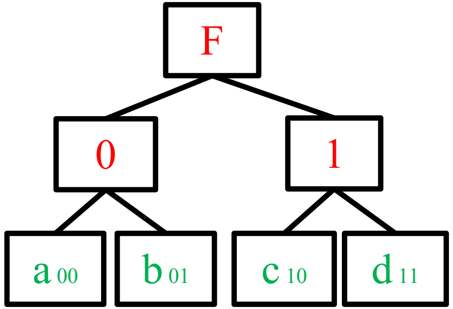


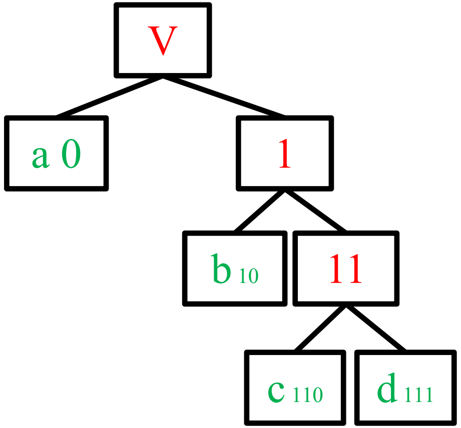
```cpp
#include "heap.h"
#include "prior.h"
using namespace std;
template <typename T> void arrayS(T* a,int n){for(int i=0;i<n;i++) cout<<a[i]<<" ";}
int main(){int iTab[16]={42,92,96,79,93,4,85,66,68,76,74,63, 39,17,71,3},it;
        cout<<"Batch initialization min-heap:\n";arrayS<int>(iTab,12);cout<<endl;
        Heap<int,IntPriorMin> aH(100,iTab,12);aH.S();cin.get();
        it=aH.remove(2);cout<<"Remove min-heap[2] => "<<it<<'\n';aH.S();cin.get();
        it=aH.remove(1);cout<<"Remove min-heap[1] => "<<it<<'\n';aH.S();cin.get();
        cout<<"Remove min-heap[-1 or 20] =>\n";aH.remove(-1);aH.remove(20);cin.get();
        it=aH.removeroot();cout<<"Remove min-heap root => "<<it<<'\n';aH.S();cin.get();
        cout<<"Insert 39 =>\n";aH.insert(39);aH.S();cin.get();
        cout<<"Insert 17 =>\n";aH.insert(17);aH.S();cin.get();
        cout<<"Clear min-heap =>\n";aH.clear();aH.S();aH.removeroot();cin.get();
        cout<<"Batch initialization min-heap:\n";arrayS<int>(iTab,16);cout<<endl;
        aH.initheap(iTab,16);aH.S();cin.get();
        cout<<"Batch initialization max-heap:\n";arrayS<int>(iTab,16);cout<<endl;
        Heap<int,IntPriorMax> bH(100,iTab,16);bH.S();cin.get();
        const char* strTab[8] = {"machine", "intelligence", "system", "automation",
                                "program", "technique", "computer", "data"};
        cout<<"Batch initialization min-heap:\n";arrayS<const char*>(strTab,8);cout<<endl;
        Heap<const char*,CharsPriorMin> strH(100,strTab,8);strH.S();cin.get();
        cout<<"Batch initialization max-heap:\n";arrayS<const char*>(strTab,8);cout<<endl;
        Heap<const char*,CharsPriorMax> strH2(100,strTab,8);strH2.S();cin.get();
        return 0;}
```

Huffman Coding Tree

coding sequence:
a a b a a b a c a d

- **Coding - variable-length vs. fixed-length**
  - fixed-length coding
    - e.g. a: 00; b: 01; c:10; d:11
    - each char is coded by 2 bits, totally 20 bits
  - *variable-length coding*
    - e.g. a: 0; b:10; c: 110; d: 111
    - chars are coded by ad hoc bits, totally 16 bits
- **Huffman coding principle**
  - statistics of chars
  - *minimum sum of weighted path lengths*
  - merge char nodes as binary tree



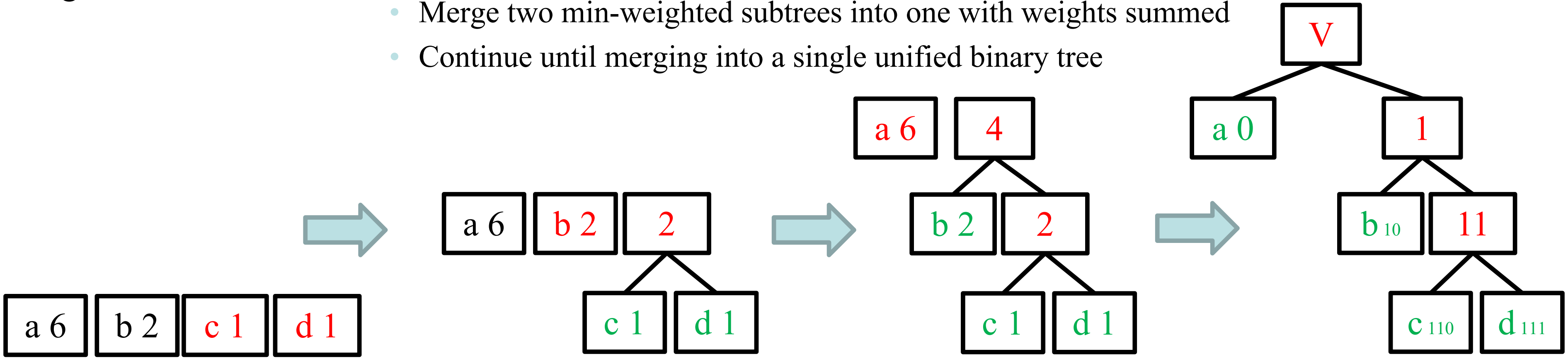


Huffman Coding Tree

- **Variable-length coding**
- **Huffman coding principle**
  - statistics of chars
  - *minimum sum of weighted path lengths*
  - merge char nodes as binary tree
    - Initialize each char as a separate weighted subtree
    - Merge two min-weighted subtrees into one with weights summed
    - Continue until merging into a single unified binary tree

coding sequence:
a a b a a b a c a d

chars	: a	b	c	d
weights	: 6	2	1	1



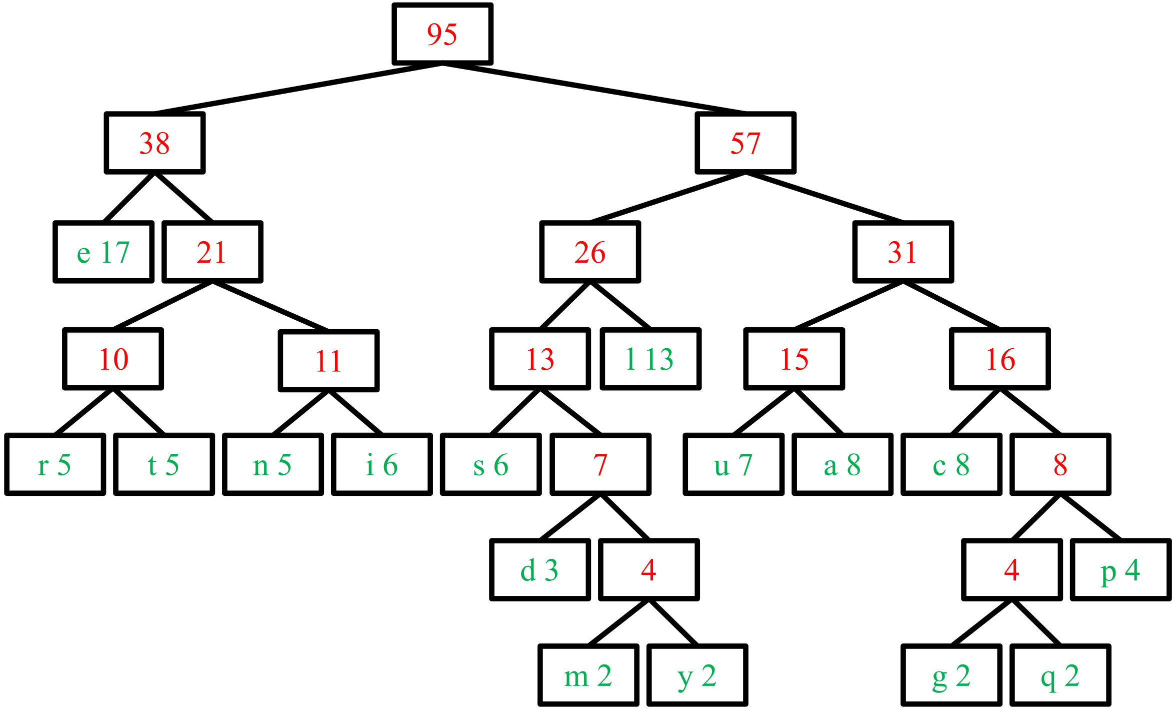
Huffman Coding Tree

- **Variable-length coding**
- **Huffman coding principle**
  - statistics of chars
  - *minimum sum of weighted path lengths*
  - merge char nodes as binary tree
    - Initialize each char as a separate weighted subtree
    - Merge two min-weighted subtrees into one with weights summed
    - Continue until merging into a single unified binary tree

Il y a un spectacle plus grand que la mer,
c'est le ciel;
il y a un spectacle plus grand que le ciel,
c'est l'intéreiur de l'âme.
蕩乎大海，壮哉其阔！
壮阔更甚大海者，乃天空；
壮阔更甚天空者，乃情怀！

Huffman Coding Tree

- **Huffman variable-length coding**



Il y a un spectacle plus grand que la mer,
c’est le ciel;
il y a un spectacle plus grand que le ciel,
c’est l’intéreiur de l’âme.

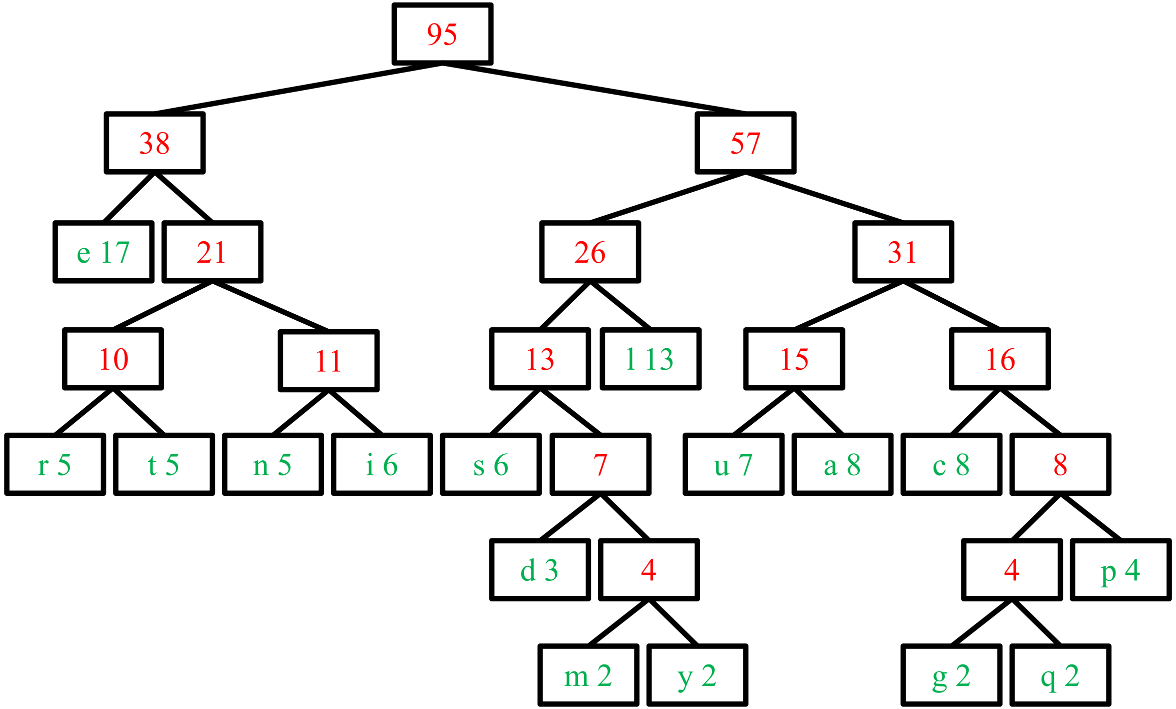
荡乎大海，壮哉其阔！
壮阔更甚大海者，乃天空；
壮阔更甚天空者，乃情怀！

n = 16 ; total weights = 95

a : 8	c : 8
d : 3	e : 17
g : 2	i : 6
l : 13	m : 2
n : 5	p : 4
q : 2	r : 5
s : 6	t : 5
u : 7	y : 2

Huffman Coding Tree

- **Huffman variable-length coding**



Il y a un spectacle plus grand que la mer,
c’est le ciel;
il y a un spectacle plus grand que le ciel,
c’est l’intéreiur de l’âme.
荡乎大海，壮哉其阔！
壮阔更甚大海者，乃天空；
壮阔更甚天空者，乃情怀！

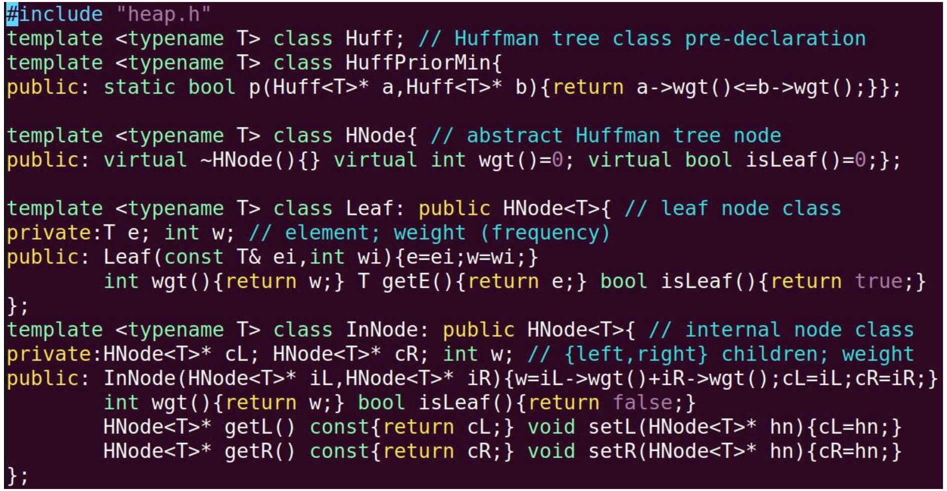
Huffman coding

a : 1101	c : 1110
d : 10010	e : 00
g : 111100	i : 0111
l : 101	m : 100110
n : 0110	p : 11111
q : 111101	r : 0100
s : 1000	t : 0101
u : 1100	y : 100111

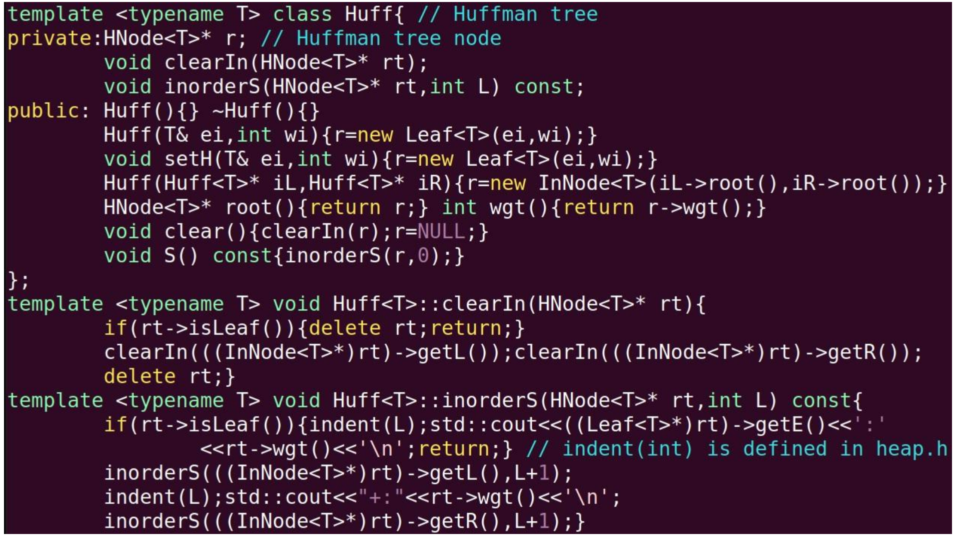
Huffman Coding Tree

- **Huffman variable-length coding**

Il y a un spectacle plus grand que la mer,
c’est le ciel;
il y a un spectacle plus grand que le ciel,
c’est l’intéreiur de l’âme.



```
#include "heap.h"
template <typename T> class Huff; // Huffman tree class pre-declaration
template <typename T> class HuffPriorMin{
public: static bool p(Huff<T>* a,Huff<T>* b){return a->wgt()<=b->wgt();}};

template <typename T> class HNode{ // abstract Huffman tree node
public: virtual ~HNode(){} virtual int wgt()=0; virtual bool isLeaf()=0;};

template <typename T> class Leaf: public HNode<T>{ // leaf node class
private:T e; int w; // element; weight (frequency)
public: Leaf(const T& ei,int wi){e=ei;w=wi;}
        int wgt(){return w;} T getE(){return e;} bool isLeaf(){return true;}
};
template <typename T> class InNode: public HNode<T>{ // internal node class
private:HNode<T>* cL; HNode<T>* cR; int w; // {left,right} children; weight
public: InNode(HNode<T>* iL,HNode<T>* iR){w=iL->wgt()+iR->wgt();cL=iL;cR=iR;}
        int wgt(){return w;} bool isLeaf(){return false;}
        HNode<T>* getL() const{return cL;} void setL(HNode<T>* hn){cL=hn;}
        HNode<T>* getR() const{return cR;} void setR(HNode<T>* hn){cR=hn;}
};
```

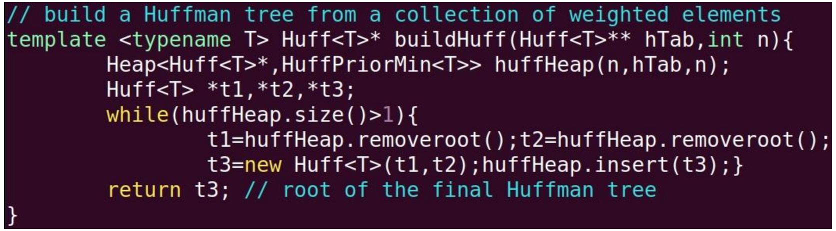


```
template <typename T> class Huff{ // Huffman tree
private:HNode<T>* r; // Huffman tree node
        void clearIn(HNode<T>* rt);
        void inorderS(HNode<T>* rt,int L) const;
public: Huff(){} ~Huff(){}
        Huff(T& ei,int wi){r=new Leaf<T>(ei,wi);}
        void setH(T& ei,int wi){r=new Leaf<T>(ei,wi);}
        Huff(Huff<T>* iL,Huff<T>* iR){r=new InNode<T>(iL->root(),iR->root());}
        HNode<T>* root(){return r;} int wgt(){return r->wgt();}
        void clear(){clearIn(r);r=NULL;}
        void S() const{inorderS(r,0);}
};
template <typename T> void Huff<T>::clearIn(HNode<T>* rt){
        if(rt->isLeaf()){delete rt;return;}
        clearIn(((InNode<T>*)rt)->getL());clearIn(((InNode<T>*)rt)->getR());
        delete rt;}
template <typename T> void Huff<T>::inorderS(HNode<T>* rt,int L) const{
        if(rt->isLeaf()){indent(L);std::cout<<((Leaf<T>*)rt)->getE()<<':'
                <<rt->wgt()<<'\n';return;} // indent(int) is defined in heap.h
        inorderS(((InNode<T>*)rt)->getL(),L+1);
        indent(L);std::cout<<"+:"<<rt->wgt()<<'\n';
        inorderS(((InNode<T>*)rt)->getR(),L+1);}
```

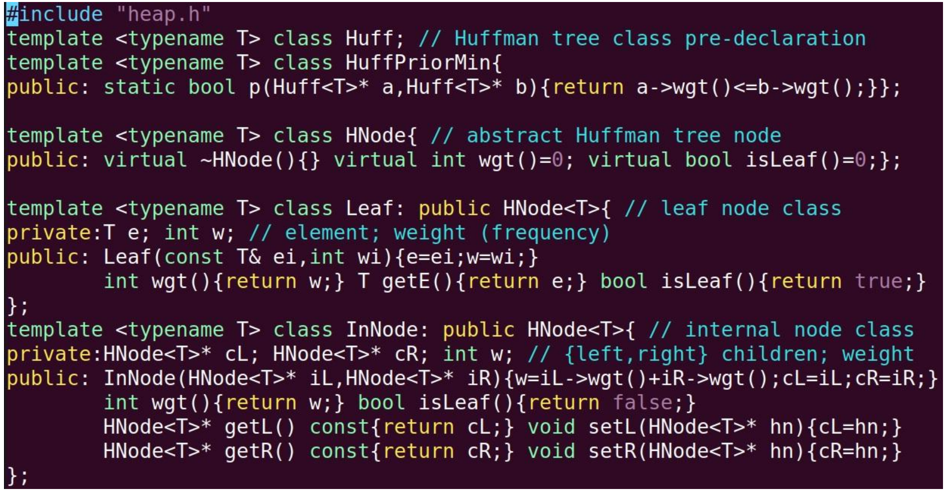
Huffman Coding Tree

- **Huffman variable-length coding**

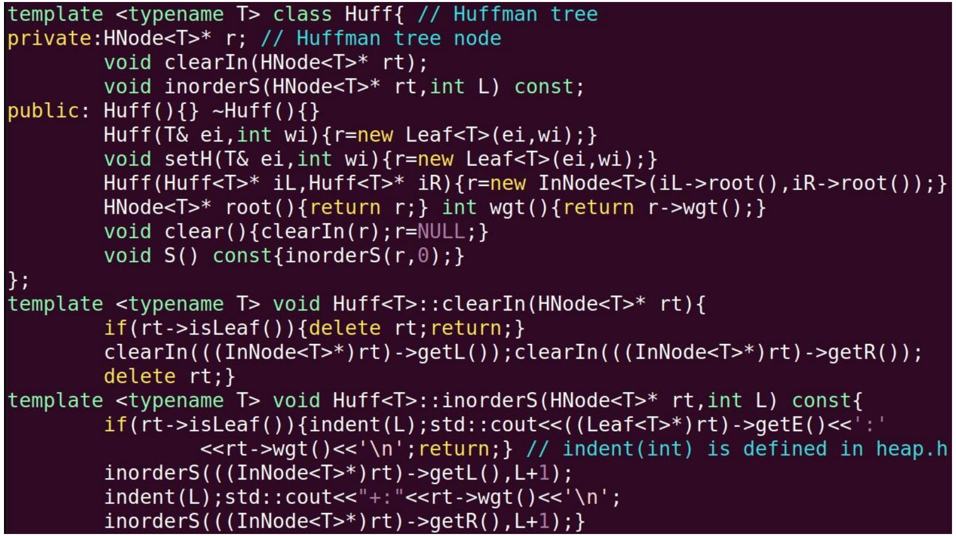
Il y a un spectacle plus grand que la mer,
c'est le ciel;
il y a un spectacle plus grand que le ciel,
c'est l'intéreiur de l'âme.



```
// build a Huffman tree from a collection of weighted elements
template <typename T> Huff<T>* buildHuff(Huff<T>** hTab,int n){
        Heap<Huff<T>*,HuffPriorMin<T>> huffHeap(n,hTab,n);
        Huff<T> *t1,*t2,*t3;
        while(huffHeap.size()>1){
                t1=huffHeap.removeroot();t2=huffHeap.removeroot();
                t3=new Huff<T>(t1,t2);huffHeap.insert(t3);}
        return t3; // root of the final Huffman tree
}
```



```
#include "heap.h"
template <typename T> class Huff; // Huffman tree class pre-declaration
template <typename T> class HuffPriorMin{
public: static bool p(Huff<T>* a,Huff<T>* b){return a->wgt()<=b->wgt();}};

template <typename T> class HNode{ // abstract Huffman tree node
public: virtual ~HNode(){} virtual int wgt()=0; virtual bool isLeaf()=0;};

template <typename T> class Leaf: public HNode<T>{ // leaf node class
private:T e; int w; // element; weight (frequency)
public: Leaf(const T& ei,int wi){e=ei;w=wi;}
        int wgt(){return w;} T getE(){return e;} bool isLeaf(){return true;}
};
template <typename T> class InNode: public HNode<T>{ // internal node class
private:HNode<T>* cL; HNode<T>* cR; int w; // {left,right} children; weight
public: InNode(HNode<T>* iL,HNode<T>* iR){w=iL->wgt()+iR->wgt();cL=iL;cR=iR;}
        int wgt(){return w;} bool isLeaf(){return false;}
        HNode<T>* getL() const{return cL;} void setL(HNode<T>* hn){cL=hn;}
        HNode<T>* getR() const{return cR;} void setR(HNode<T>* hn){cR=hn;}
};
```



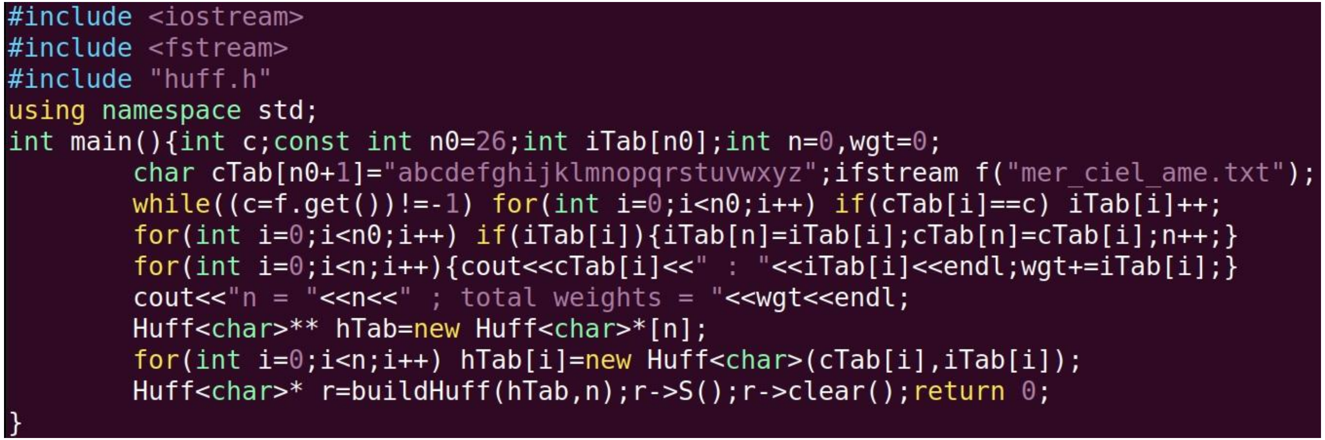
```
template <typename T> class Huff{ // Huffman tree
private:HNode<T>* r; // Huffman tree node
        void clearIn(HNode<T>* rt);
        void inorderS(HNode<T>* rt,int L) const;
public: Huff(){} ~Huff(){}
        Huff(T& ei,int wi){r=new Leaf<T>(ei,wi);}
        void setH(T& ei,int wi){r=new Leaf<T>(ei,wi);}
        Huff(Huff<T>* iL,Huff<T>* iR){r=new InNode<T>(iL->root(),iR->root());}
        HNode<T>* root(){return r;} int wgt(){return r->wgt();}
        void clear(){clearIn(r);r=NULL;}
        void S() const{inorderS(r,0);}
};
template <typename T> void Huff<T>::clearIn(HNode<T>* rt){
        if(rt->isLeaf()){delete rt;return;}
        clearIn(((InNode<T>*)rt)->getL());clearIn(((InNode<T>*)rt)->getR());
        delete rt;}
template <typename T> void Huff<T>::inorderS(HNode<T>* rt,int L) const{
        if(rt->isLeaf()){indent(L);std::cout<<((Leaf<T>*)rt)->getE()<<':'
                <<rt->wgt()<<'\n';return;} // indent(int) is defined in heap.h
        inorderS(((InNode<T>*)rt)->getL(),L+1);
        indent(L);std::cout<<"+:"<<rt->wgt()<<'\n';
        inorderS(((InNode<T>*)rt)->getR(),L+1);}
```

Huffman Coding Tree

```
        e:17
    +:38
                r:5
            +:10
                t:5
        +:21
                n:5
            +:11
                i:6
+:95
                s:6
            +:13
                    d:3
                +:7
                        m:2
                    +:4
                        y:2
        +:26
            l:13
    +:57
                u:7
            +:15
                a:8
        +:31
                c:8
            +:16
                        g:2
                    +:4
                        q:2
                +:8
                    p:4
```

- **Huffman variable-length coding**

Il y a un spectacle plus grand que la mer,
c'est le ciel;
il y a un spectacle plus grand que le ciel,
c'est l'intéreiur de l'âme.
荡乎大海，壮哉其阔！
壮阔更甚大海者，乃天空；
壮阔更甚天空者，乃情怀！



```
#include <iostream>
#include <fstream>
#include "huff.h"
using namespace std;
int main(){int c;const int n0=26;int iTab[n0];int n=0,wgt=0;
        char cTab[n0+1]="abcdefghijklmnopqrstuvwxyz";ifstream f("mer_ciel_ame.txt");
        while((c=f.get())!=-1) for(int i=0;i<n0;i++) if(cTab[i]==c) iTab[i]++;
        for(int i=0;i<n0;i++) if(iTab[i]){iTab[n]=iTab[i];cTab[n]=cTab[i];n++;}
        for(int i=0;i<n;i++){cout<<cTab[i]<<" : "<<iTab[i]<<endl;wgt+=iTab[i];}
        cout<<"n = "<<n<<" ; total weights = "<<wgt<<endl;
        Huff<char>** hTab=new Huff<char>*[n];
        for(int i=0;i<n;i++) hTab[i]=new Huff<char>(cTab[i],iTab[i]);
        Huff<char>* r=buildHuff(hTab,n);r->S();r->clear();return 0;
}
```

THANK YOU

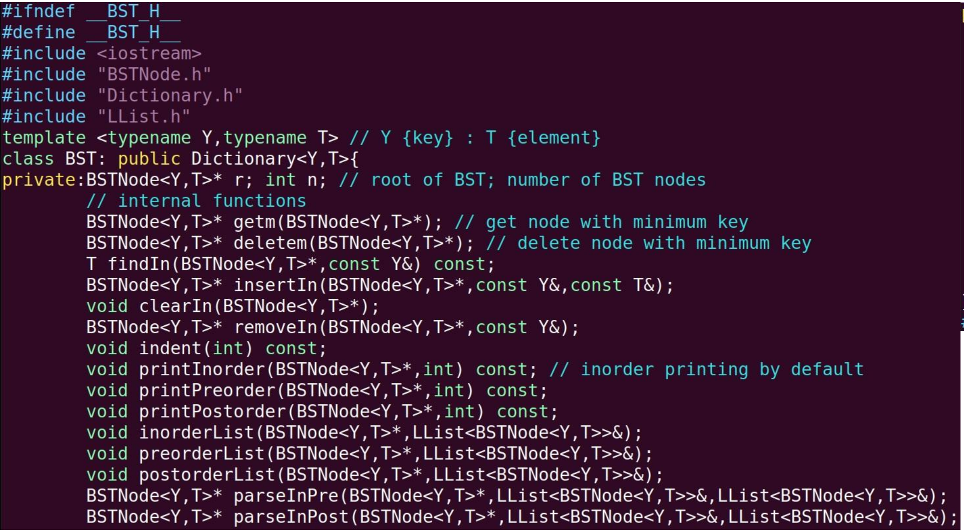
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Course Project 1

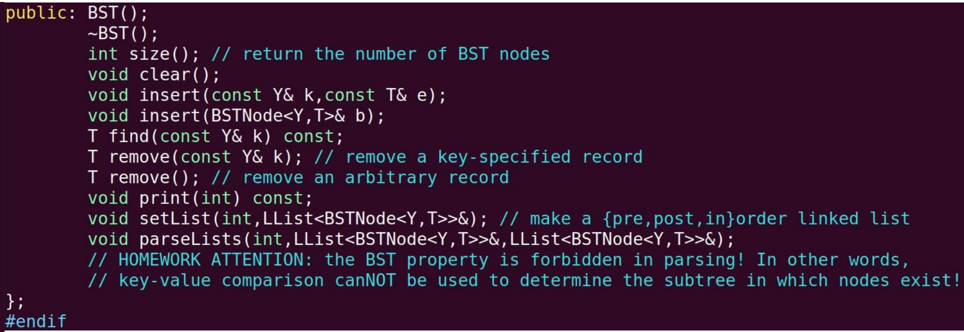
- **Binary search tree**
  - complete implementation code of the BST class
    - **complete BST member functions in the public part**, whereas internal functions in the private part are only for guiding purpose but not mandatory so.
    - just put implementation code **in the header file BST.h** (for your convenience)
    - accomplishment of each member function will also be evaluated individually
  - verify correctness of your implementation code via the given main code
    - just copy & **leave the main code as it is**; do NOT touch it
  - realize functions of parsing {in,pre}order lists & parsing {in,post}order lists
    - this serves as highlight of course project 1
    - recover the BST by parsing its associated {in,pre}order lists
    - recover the BST by parsing its associated {in,post}order lists
    - **the BST property is fobidden in parsing**! in other words, canNOT use key-value comparison to determine the subtree where certain node exists.

Course Project 1

- **Binary search tree**
  - complete implementation code of the BST class
  - verify correctness of your implementation code via the given main code
  - realize functions of parsing {in,pre}order lists & parsing {in,post}order lists



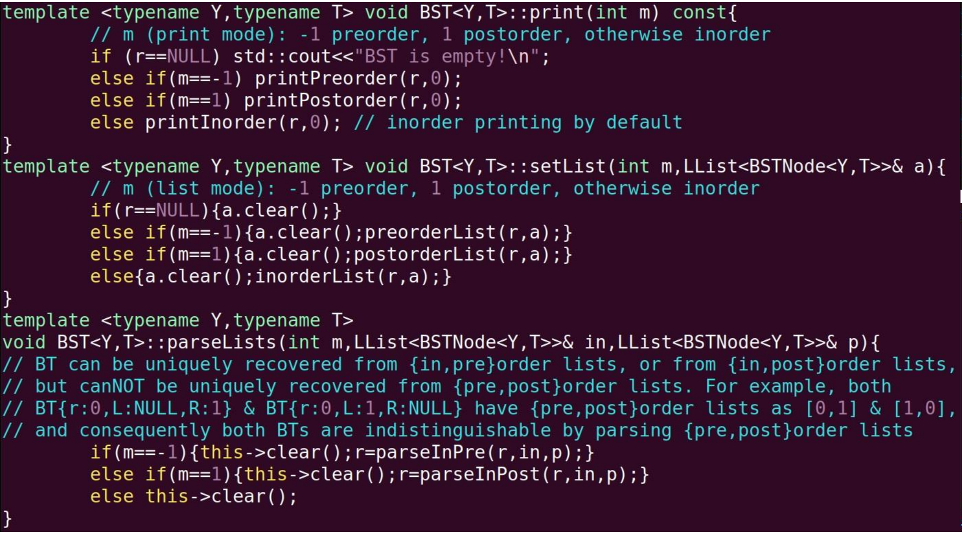
```cpp
#ifndef __BST_H__
#define __BST_H__
#include <iostream>
#include "BSTNode.h"
#include "Dictionary.h"
#include "LList.h"
template <typename Y,typename T> // Y {key} : T {element}
class BST: public Dictionary<Y,T>{
private:BSTNode<Y,T>* r; int n; // root of BST; number of BST nodes
        // internal functions
        BSTNode<Y,T>* getm(BSTNode<Y,T>*); // get node with minimum key
        BSTNode<Y,T>* deletem(BSTNode<Y,T>*); // delete node with minimum key
        T findIn(BSTNode<Y,T>*,const Y&) const;
        BSTNode<Y,T>* insertIn(BSTNode<Y,T>*,const Y&,const T&);
        void clearIn(BSTNode<Y,T>*);
        BSTNode<Y,T>* removeIn(BSTNode<Y,T>*,const Y&);
        void indent(int) const;
        void printInorder(BSTNode<Y,T>*,int) const; // inorder printing by default
        void printPreorder(BSTNode<Y,T>*,int) const;
        void printPostorder(BSTNode<Y,T>*,int) const;
        void inorderList(BSTNode<Y,T>*,LList<BSTNode<Y,T>>&);
        void preorderList(BSTNode<Y,T>*,LList<BSTNode<Y,T>>&);
        void postorderList(BSTNode<Y,T>*,LList<BSTNode<Y,T>>&);
        BSTNode<Y,T>* parseInPre(BSTNode<Y,T>*,LList<BSTNode<Y,T>>&,LList<BSTNode<Y,T>>&);
        BSTNode<Y,T>* parseInPost(BSTNode<Y,T>*,LList<BSTNode<Y,T>>&,LList<BSTNode<Y,T>>&);
```



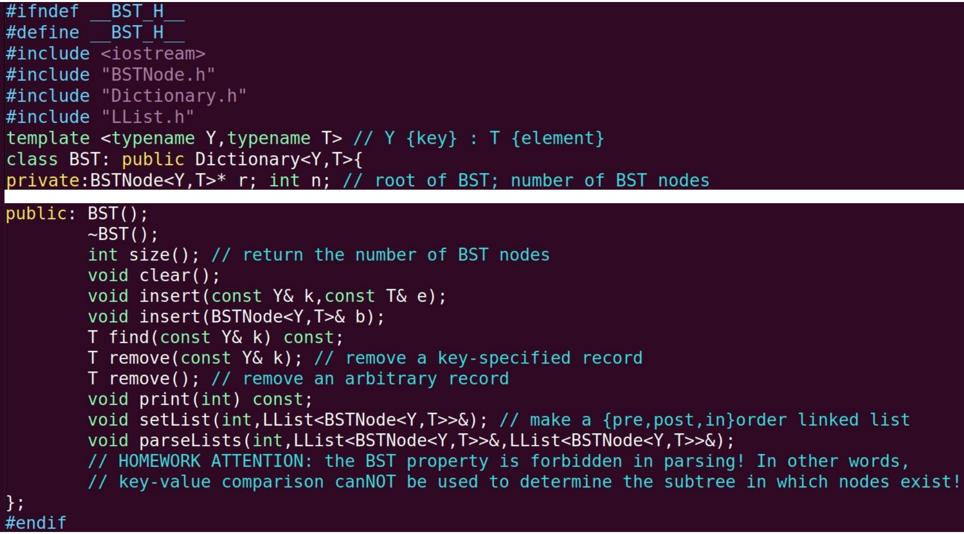
```cpp
public: BST();
        ~BST();
        int size(); // return the number of BST nodes
        void clear();
        void insert(const Y& k,const T& e);
        void insert(BSTNode<Y,T>& b);
        T find(const Y& k) const;
        T remove(const Y& k); // remove a key-specified record
        T remove(); // remove an arbitrary record
        void print(int) const;
        void setList(int,LList<BSTNode<Y,T>>&); // make a {pre,post,in}order linked list
        void parseLists(int,LList<BSTNode<Y,T>>&,LList<BSTNode<Y,T>>&);
        // HOMEWORK ATTENTION: the BST property is forbidden in parsing! In other words,
        // key-value comparison canNOT be used to determine the subtree in which nodes exist!
};
#endif
```

Course Project 1

- **Binary search tree**
  - complete implementation code of the BST class
  - verify correctness of your implementation code via the given main code
  - realize functions of parsing {in,pre}order lists & parsing {in,post}order lists



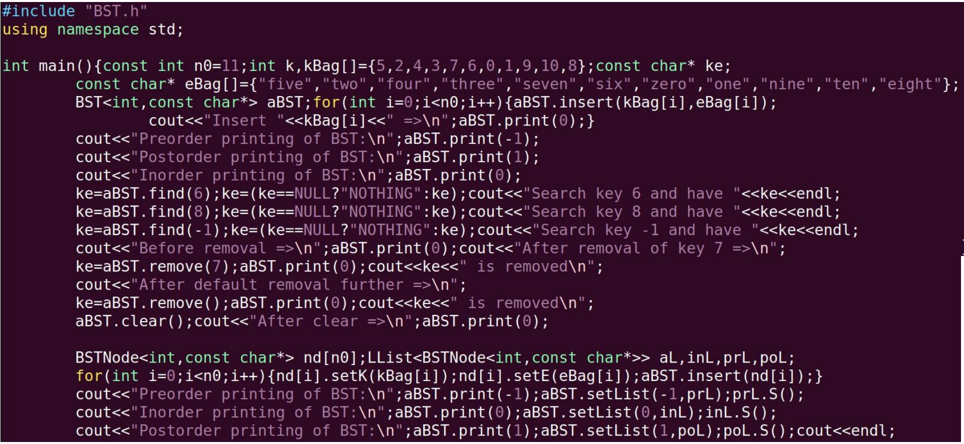
```
template <typename Y,typename T> void BST<Y,T>::print(int m) const{
        // m (print mode): -1 preorder, 1 postorder, otherwise inorder
        if (r==NULL) std::cout<<"BST is empty!\n";
        else if(m==-1) printPreorder(r,0);
        else if(m==1) printPostorder(r,0);
        else printInorder(r,0); // inorder printing by default
}
template <typename Y,typename T> void BST<Y,T>::setList(int m,LList<BSTNode<Y,T>>& a){
        // m (list mode): -1 preorder, 1 postorder, otherwise inorder
        if(r==NULL){a.clear();}
        else if(m==-1){a.clear();preorderList(r,a);}
        else if(m==1){a.clear();postorderList(r,a);}
        else{a.clear();inorderList(r,a);}
}
template <typename Y,typename T>
void BST<Y,T>::parseLists(int m,LList<BSTNode<Y,T>>& in,LList<BSTNode<Y,T>>& p){
// BT can be uniquely recovered from {in,pre}order lists, or from {in,post}order lists,
// but canNOT be uniquely recovered from {pre,post}order lists. For example, both
// BT{r:0,L:NULL,R:1} & BT{r:0,L:1,R:NULL} have {pre,post}order lists as [0,1] & [1,0],
// and consequently both BTs are indistinguishable by parsing {pre,post}order lists
        if(m==-1){this->clear();r=parseInPre(r,in,p);}
        else if(m==1){this->clear();r=parseInPost(r,in,p);}
        else this->clear();
}
```



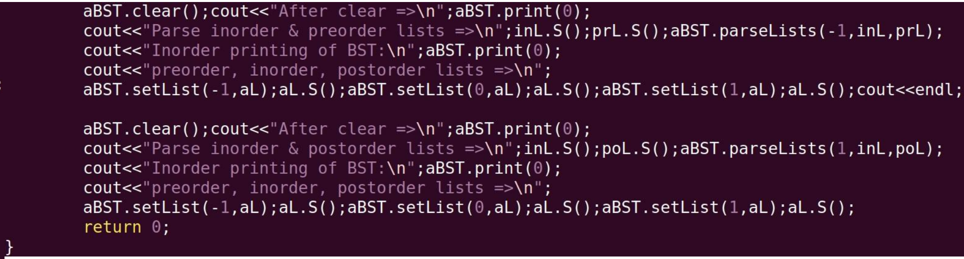
```
#ifndef __BST_H__
#define __BST_H__
#include <iostream>
#include "BSTNode.h"
#include "Dictionary.h"
#include "LList.h"
template <typename Y,typename T> // Y {key} : T {element}
class BST: public Dictionary<Y,T>{
private:BSTNode<Y,T>* r; int n; // root of BST; number of BST nodes

public: BST();
        ~BST();
        int size(); // return the number of BST nodes
        void clear();
        void insert(const Y& k,const T& e);
        void insert(BSTNode<Y,T>& b);
        T find(const Y& k) const;
        T remove(const Y& k); // remove a key-specified record
        T remove(); // remove an arbitrary record
        void print(int) const;
        void setList(int,LList<BSTNode<Y,T>>&); // make a {pre,post,in}order linked list
        void parseLists(int,LList<BSTNode<Y,T>>&,LList<BSTNode<Y,T>>&);
        // HOMEWORK ATTENTION: the BST property is forbidden in parsing! In other words,
        // key-value comparison canNOT be used to determine the subtree in which nodes exist!
};
#endif
```

Course Project 1

- **Binary search tree**
  - complete implementation code of the BST class
  - verify correctness of your implementation code via the given main code
  - realize functions of parsing {in,pre}order lists & parsing {in,post}order lists



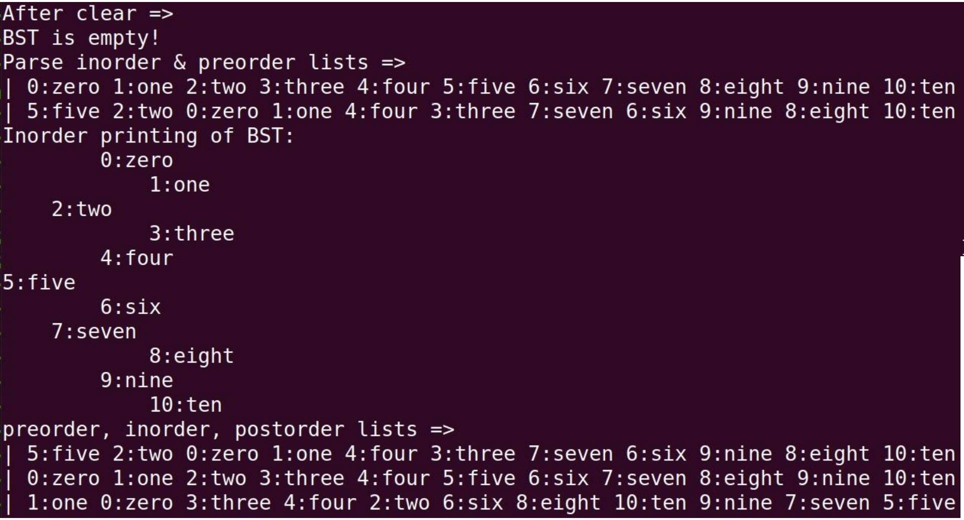
```
#include "BST.h"
using namespace std;

int main(){const int n0=11;int k,kBag[]={5,2,4,3,7,6,0,1,9,10,8};const char* ke;
        const char* eBag[]={"five","two","four","three","seven","six","zero","one","nine","ten","eight"};
        BST<int,const char*> aBST;for(int i=0;i<n0;i++){aBST.insert(kBag[i],eBag[i]);
                cout<<"Insert "<<kBag[i]<<" =>\n";aBST.print(0);}
        cout<<"Preorder printing of BST:\n";aBST.print(-1);
        cout<<"Postorder printing of BST:\n";aBST.print(1);
        cout<<"Inorder printing of BST:\n";aBST.print(0);
        ke=aBST.find(6);ke=(ke==NULL?"NOTHING":ke);cout<<"Search key 6 and have "<<ke<<endl;
        ke=aBST.find(8);ke=(ke==NULL?"NOTHING":ke);cout<<"Search key 8 and have "<<ke<<endl;
        ke=aBST.find(-1);ke=(ke==NULL?"NOTHING":ke);cout<<"Search key -1 and have "<<ke<<endl;
        cout<<"Before removal =>\n";aBST.print(0);cout<<"After removal of key 7 =>\n";
        ke=aBST.remove(7);aBST.print(0);cout<<ke<<" is removed\n";
        cout<<"After default removal further =>\n";
        ke=aBST.remove();aBST.print(0);cout<<ke<<" is removed\n";
        aBST.clear();cout<<"After clear =>\n";aBST.print(0);

        BSTNode<int,const char*> nd[n0];LList<BSTNode<int,const char*>> aL,inL,prL,poL;
        for(int i=0;i<n0;i++){nd[i].setK(kBag[i]);nd[i].setE(eBag[i]);aBST.insert(nd[i]);}
        cout<<"Preorder printing of BST:\n";aBST.print(-1);aBST.setList(-1,prL);prL.S();
        cout<<"Inorder printing of BST:\n";aBST.print(0);aBST.setList(0,inL);inL.S();
        cout<<"Postorder printing of BST:\n";aBST.print(1);aBST.setList(1,poL);poL.S();cout<<endl;
```



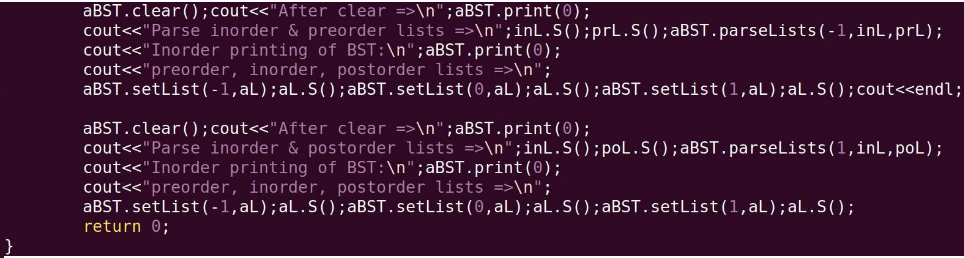
```
        aBST.clear();cout<<"After clear =>\n";aBST.print(0);
        cout<<"Parse inorder & preorder lists =>\n";inL.S();prL.S();aBST.parseLists(-1,inL,prL);
        cout<<"Inorder printing of BST:\n";aBST.print(0);
        cout<<"preorder, inorder, postorder lists =>\n";
        aBST.setList(-1,aL);aL.S();aBST.setList(0,aL);aL.S();aBST.setList(1,aL);aL.S();cout<<endl;

        aBST.clear();cout<<"After clear =>\n";aBST.print(0);
        cout<<"Parse inorder & postorder lists =>\n";inL.S();poL.S();aBST.parseLists(1,inL,poL);
        cout<<"Inorder printing of BST:\n";aBST.print(0);
        cout<<"preorder, inorder, postorder lists =>\n";
        aBST.setList(-1,aL);aL.S();aBST.setList(0,aL);aL.S();aBST.setList(1,aL);aL.S();
        return 0;
}
```

Course Project 1

- **Binary search tree**
  - complete implementation code of the BST class
  - verify correctness of your implementation code via the given main code
  - realize functions of parsing {in,pre}order lists & parsing {in,post}order lists



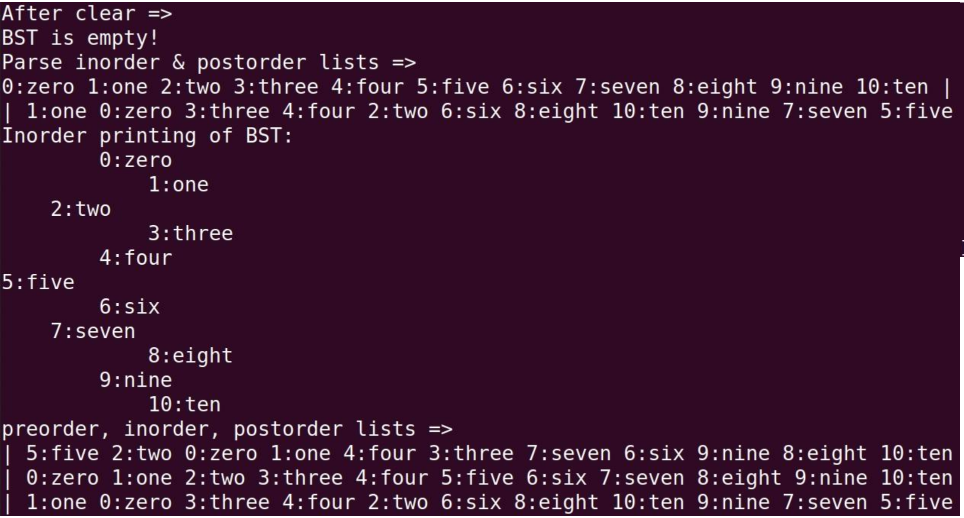
```
After clear =>
BST is empty!
Parse inorder & preorder lists =>
| 0:zero 1:one 2:two 3:three 4:four 5:five 6:six 7:seven 8:eight 9:nine 10:ten
| 5:five 2:two 0:zero 1:one 4:four 3:three 7:seven 6:six 9:nine 8:eight 10:ten
Inorder printing of BST:
        0:zero
            1:one
    2:two
            3:three
        4:four
5:five
        6:six
    7:seven
            8:eight
        9:nine
            10:ten
preorder, inorder, postorder lists =>
| 5:five 2:two 0:zero 1:one 4:four 3:three 7:seven 6:six 9:nine 8:eight 10:ten
| 0:zero 1:one 2:two 3:three 4:four 5:five 6:six 7:seven 8:eight 9:nine 10:ten
| 1:one 0:zero 3:three 4:four 2:two 6:six 8:eight 10:ten 9:nine 7:seven 5:five
```



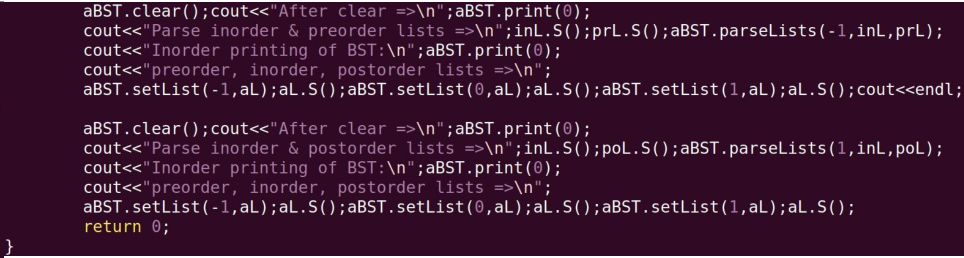
```
        aBST.clear();cout<<"After clear =>\n";aBST.print(0);
        cout<<"Parse inorder & preorder lists =>\n";inL.S();prL.S();aBST.parseLists(-1,inL,prL);
        cout<<"Inorder printing of BST:\n";aBST.print(0);
        cout<<"preorder, inorder, postorder lists =>\n";
        aBST.setList(-1,aL);aL.S();aBST.setList(0,aL);aL.S();aBST.setList(1,aL);aL.S();cout<<endl;

        aBST.clear();cout<<"After clear =>\n";aBST.print(0);
        cout<<"Parse inorder & postorder lists =>\n";inL.S();poL.S();aBST.parseLists(1,inL,poL);
        cout<<"Inorder printing of BST:\n";aBST.print(0);
        cout<<"preorder, inorder, postorder lists =>\n";
        aBST.setList(-1,aL);aL.S();aBST.setList(0,aL);aL.S();aBST.setList(1,aL);aL.S();
        return 0;
}
```

Course Project 1

- **Binary search tree**
  - complete implementation code of the BST class
  - verify correctness of your implementation code via the given main code
  - realize functions of parsing {in,pre}order lists & parsing {in,post}order lists



```
After clear =>
BST is empty!
Parse inorder & postorder lists =>
0:zero 1:one 2:two 3:three 4:four 5:five 6:six 7:seven 8:eight 9:nine 10:ten |
| 1:one 0:zero 3:three 4:four 2:two 6:six 8:eight 10:ten 9:nine 7:seven 5:five
Inorder printing of BST:
    0:zero
      1:one
  2:two
      3:three
    4:four
5:five
    6:six
  7:seven
      8:eight
    9:nine
      10:ten
preorder, inorder, postorder lists =>
| 5:five 2:two 0:zero 1:one 4:four 3:three 7:seven 6:six 9:nine 8:eight 10:ten
| 0:zero 1:one 2:two 3:three 4:four 5:five 6:six 7:seven 8:eight 9:nine 10:ten
| 1:one 0:zero 3:three 4:four 2:two 6:six 8:eight 10:ten 9:nine 7:seven 5:five
```



```
    aBST.clear();cout<<"After clear =>\n";aBST.print(0);
    cout<<"Parse inorder & preorder lists =>\n";inL.S();prL.S();aBST.parseLists(-1,inL,prL);
    cout<<"Inorder printing of BST:\n";aBST.print(0);
    cout<<"preorder, inorder, postorder lists =>\n";
    aBST.setList(-1,aL);aL.S();aBST.setList(0,aL);aL.S();aBST.setList(1,aL);aL.S();cout<<endl;

    aBST.clear();cout<<"After clear =>\n";aBST.print(0);
    cout<<"Parse inorder & postorder lists =>\n";inL.S();poL.S();aBST.parseLists(1,inL,poL);
    cout<<"Inorder printing of BST:\n";aBST.print(0);
    cout<<"preorder, inorder, postorder lists =>\n";
    aBST.setList(-1,aL);aL.S();aBST.setList(0,aL);aL.S();aBST.setList(1,aL);aL.S();
    return 0;
}
```

THANK YOU

上海交通大學

SHANGHAI JIAO TONG UNIVERSITY