Algoritmi e Strutture Dati

Lezione 4

www.iet.unipi.it/a.virdis

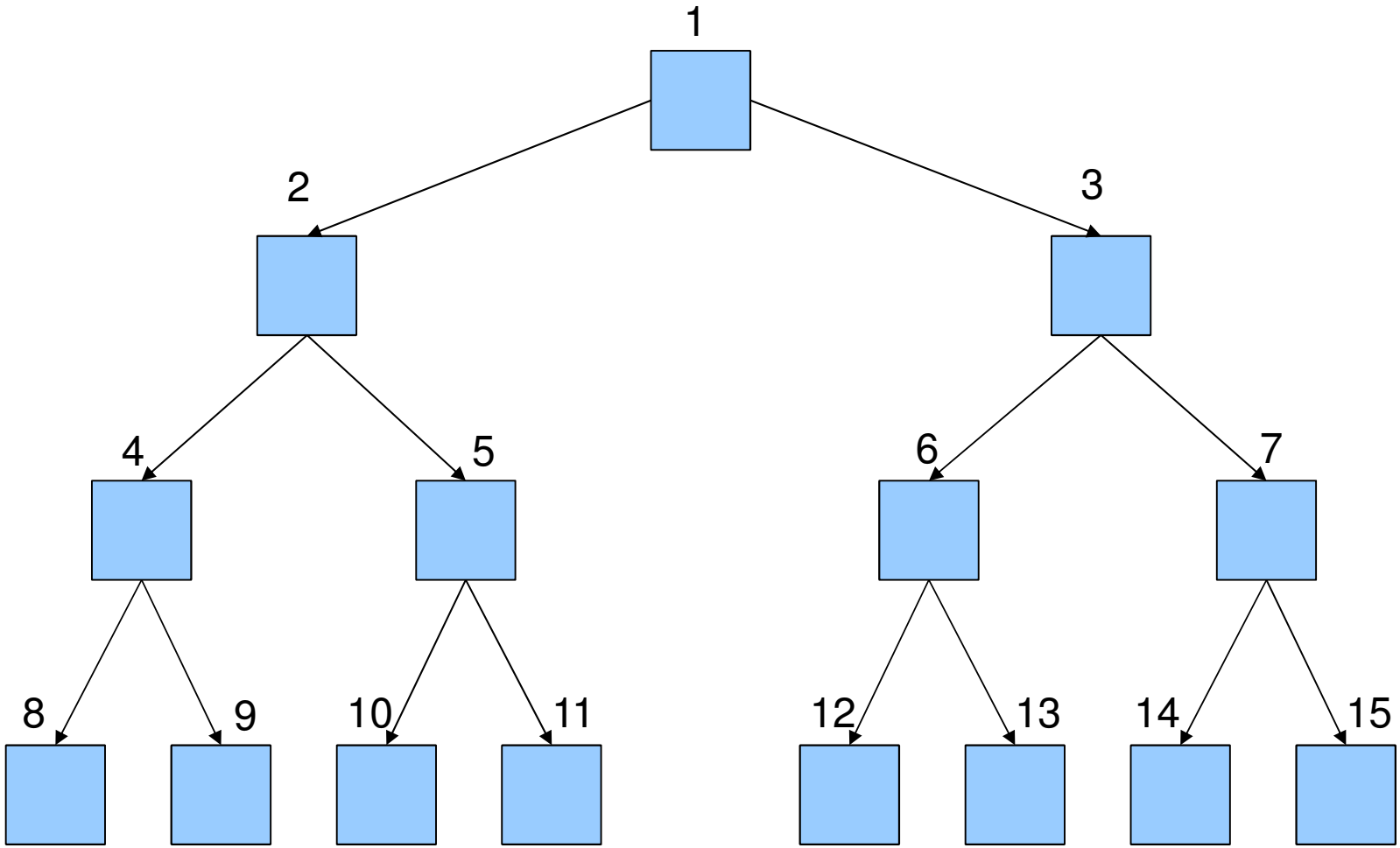
Antonio Virdis

a.virdis@iet.unipi.it

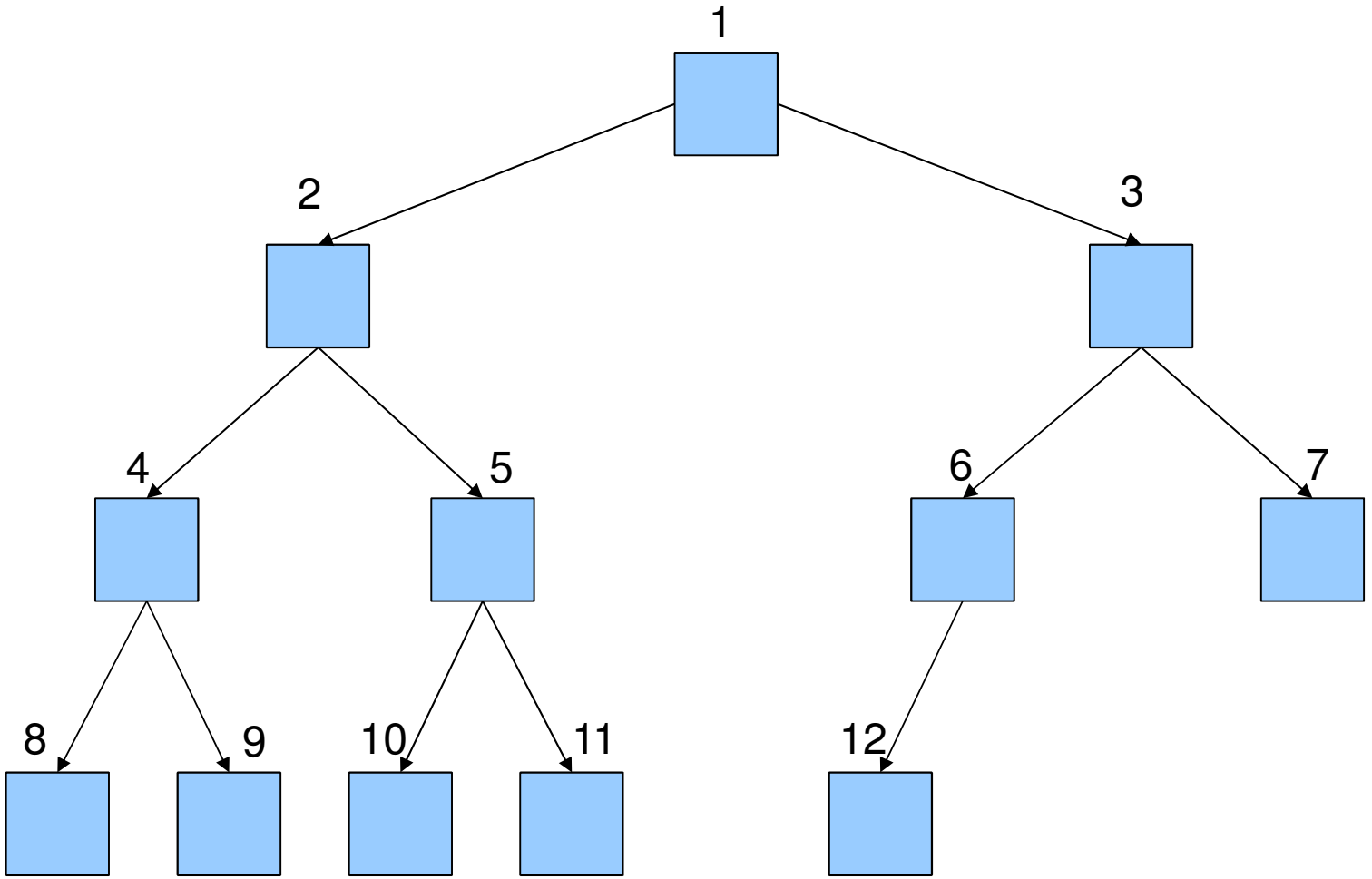
Sommario

- Heap
- Ordinamento tramite Heap
- Soluzioni
- Esercizi

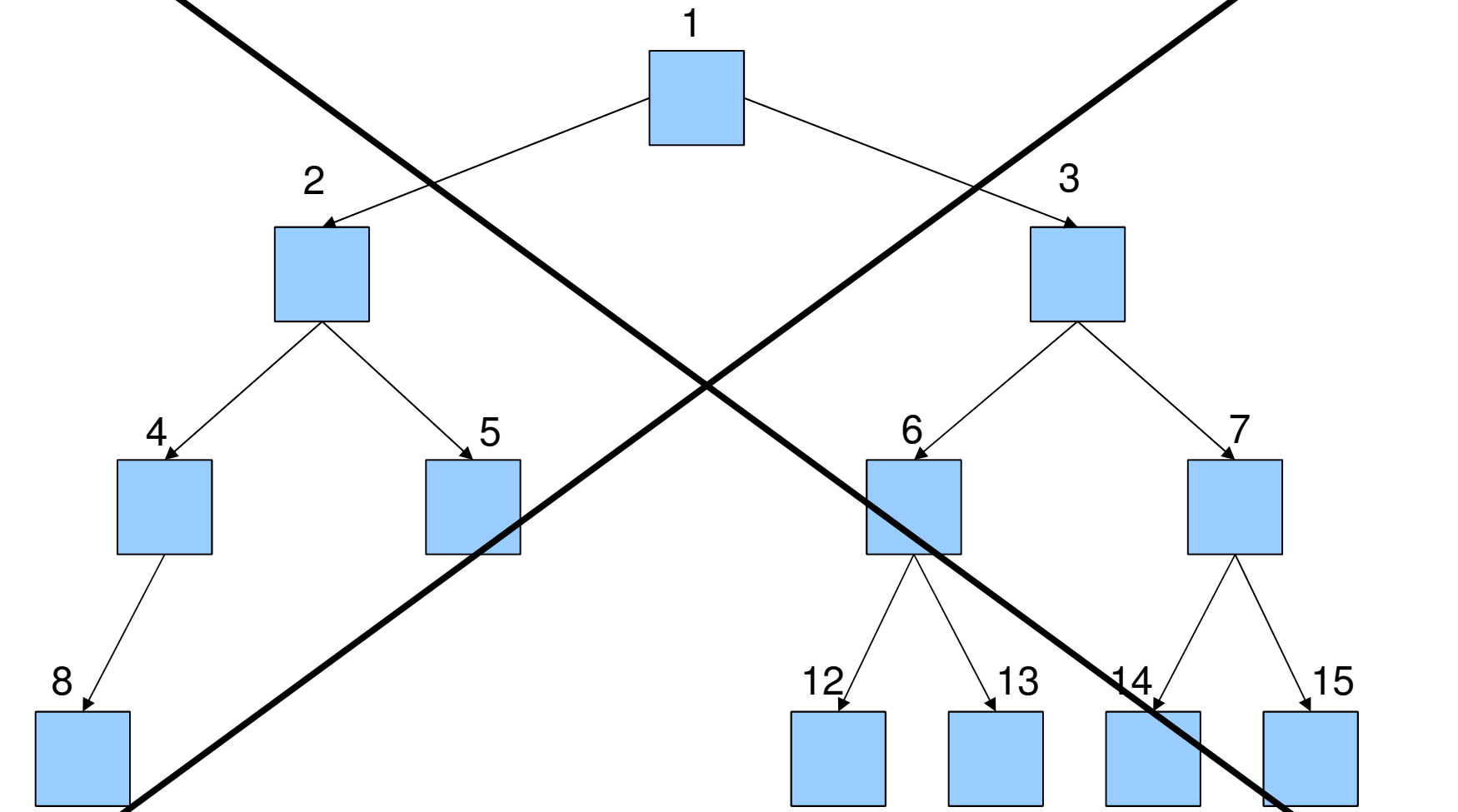
heap



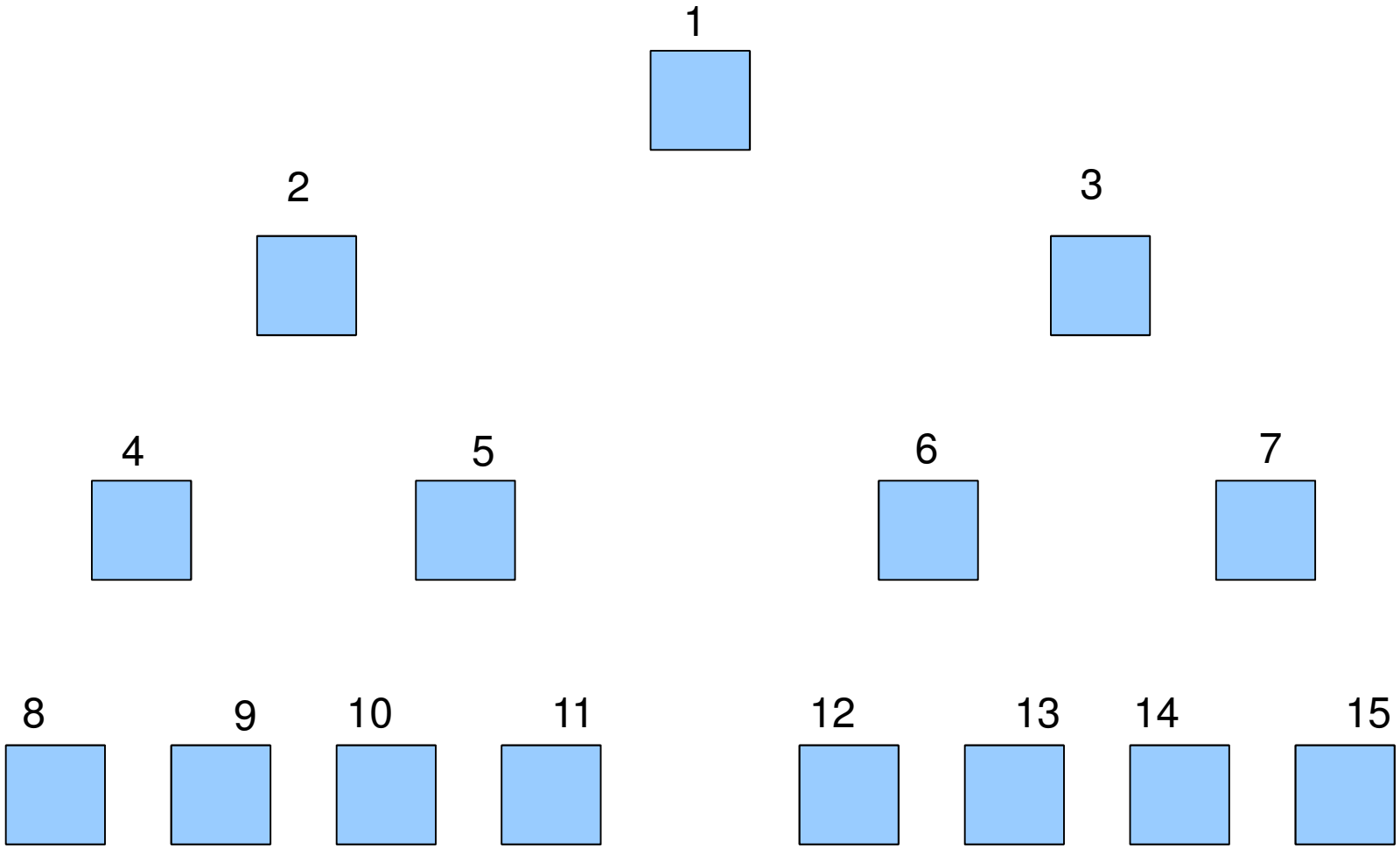
heap



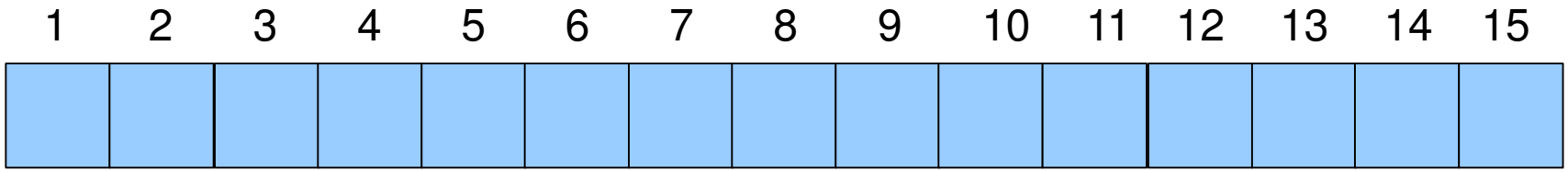
heap



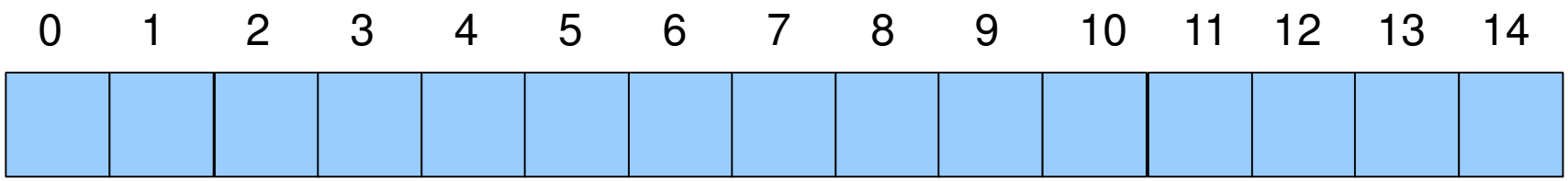
heap



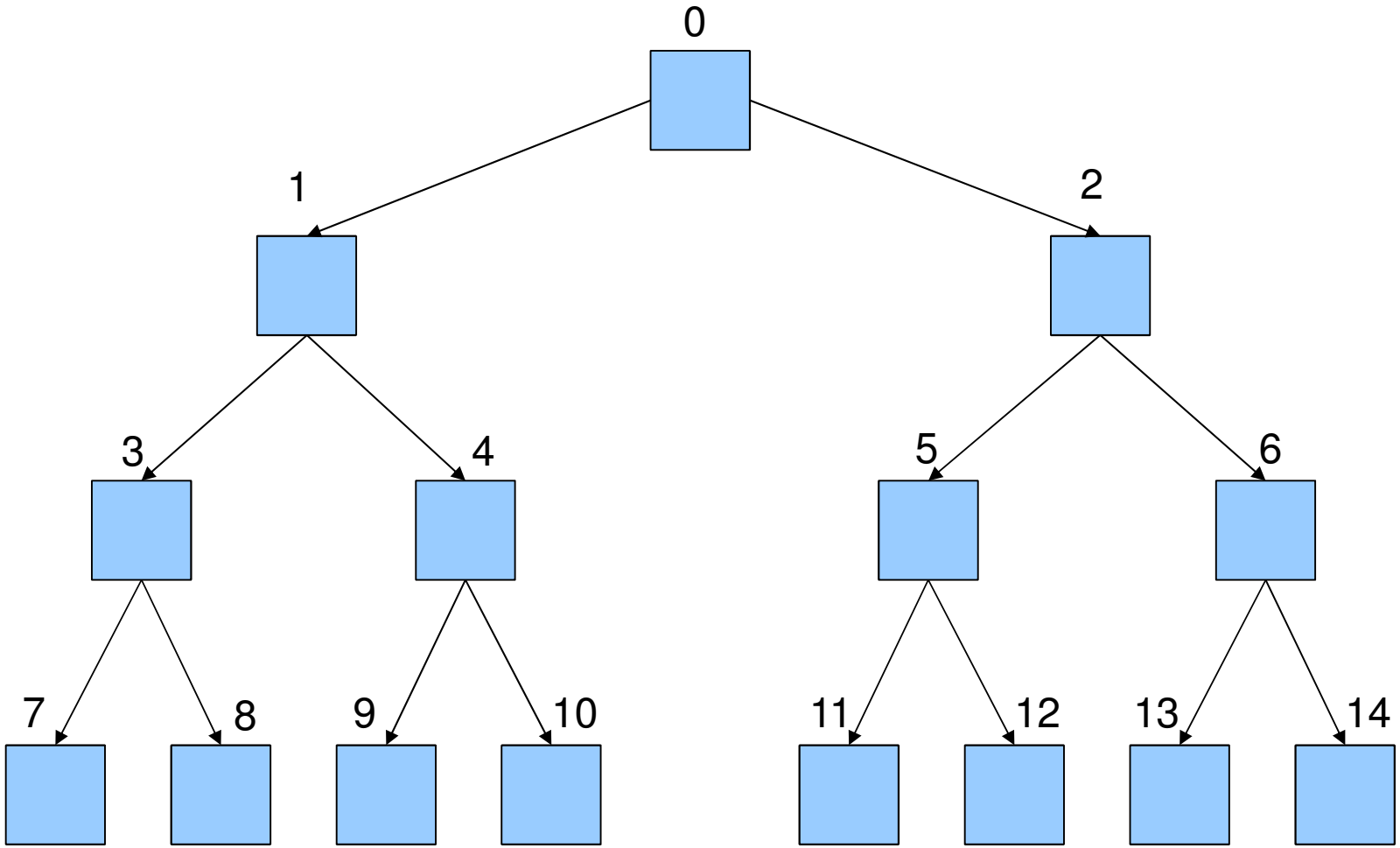
heap



heap



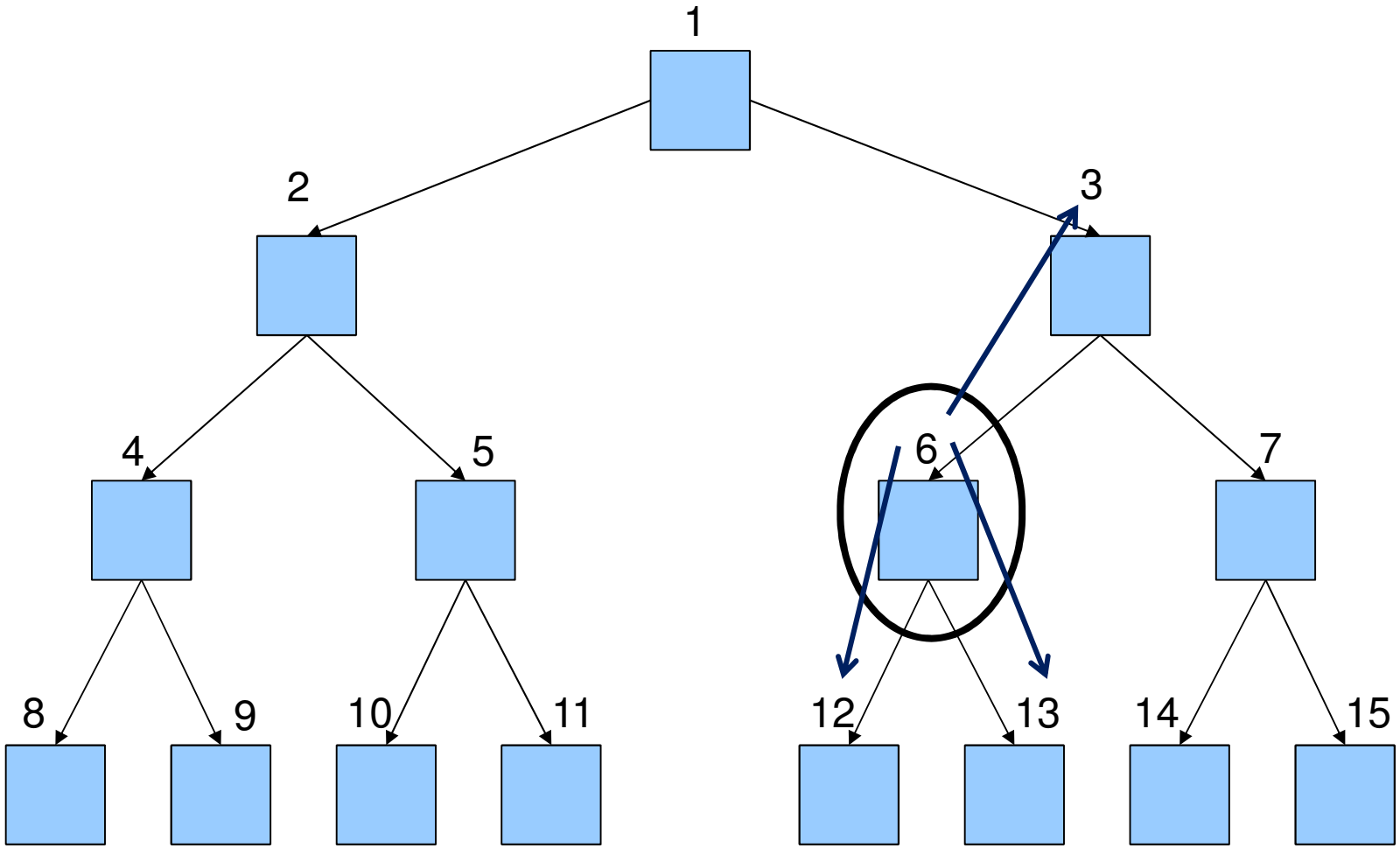
heap



heap

```
1   class Heap
2   {
3       std::vector<int> data_;
4
5       int length_;     // lunghezza array
6       int size_;   // dimensione Heap
7
8   public:
9       Heap() {};
10
11      void fill( int l );
12      void printVector();
13
14      ...
15  }
```

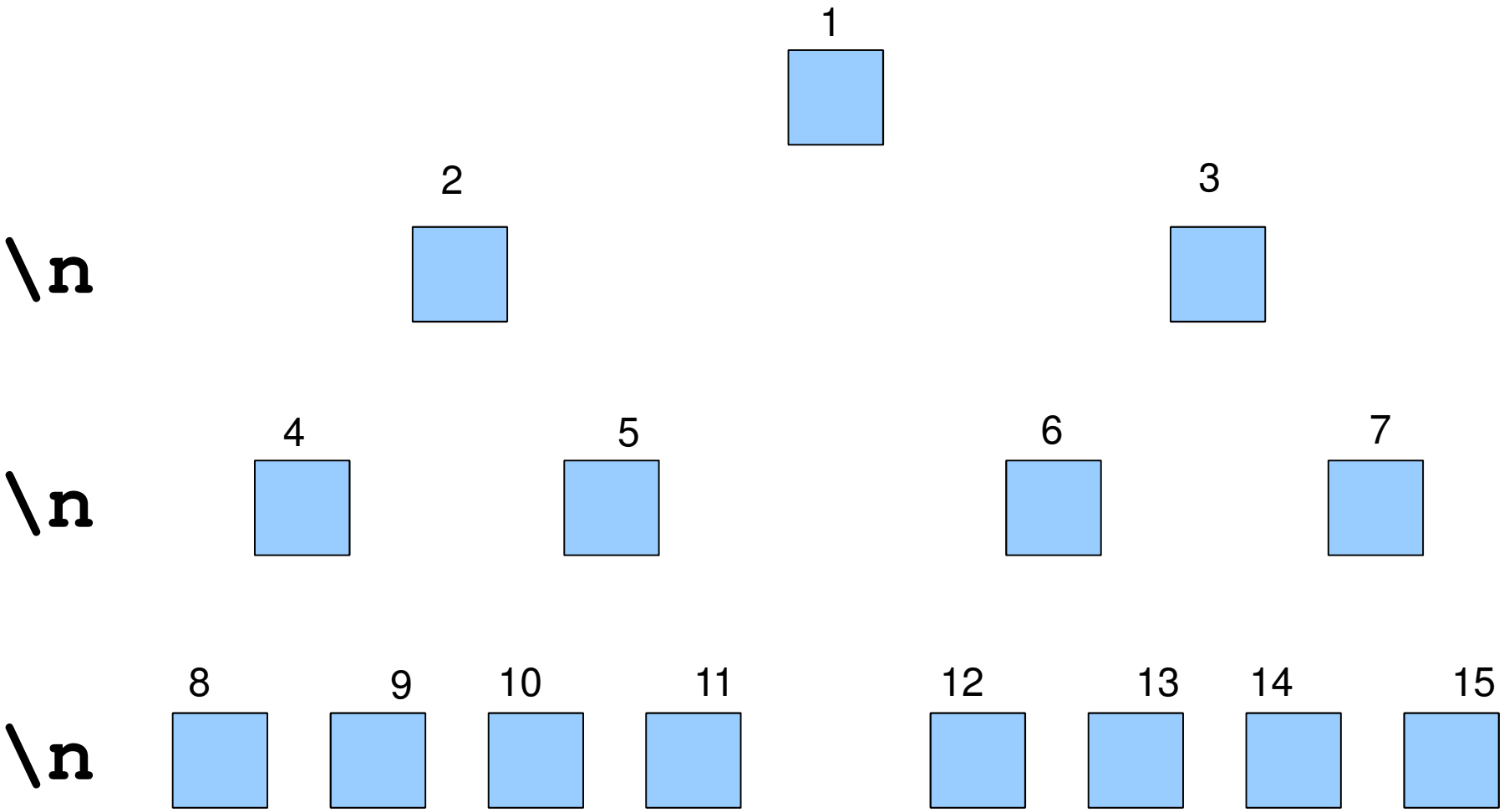
heap



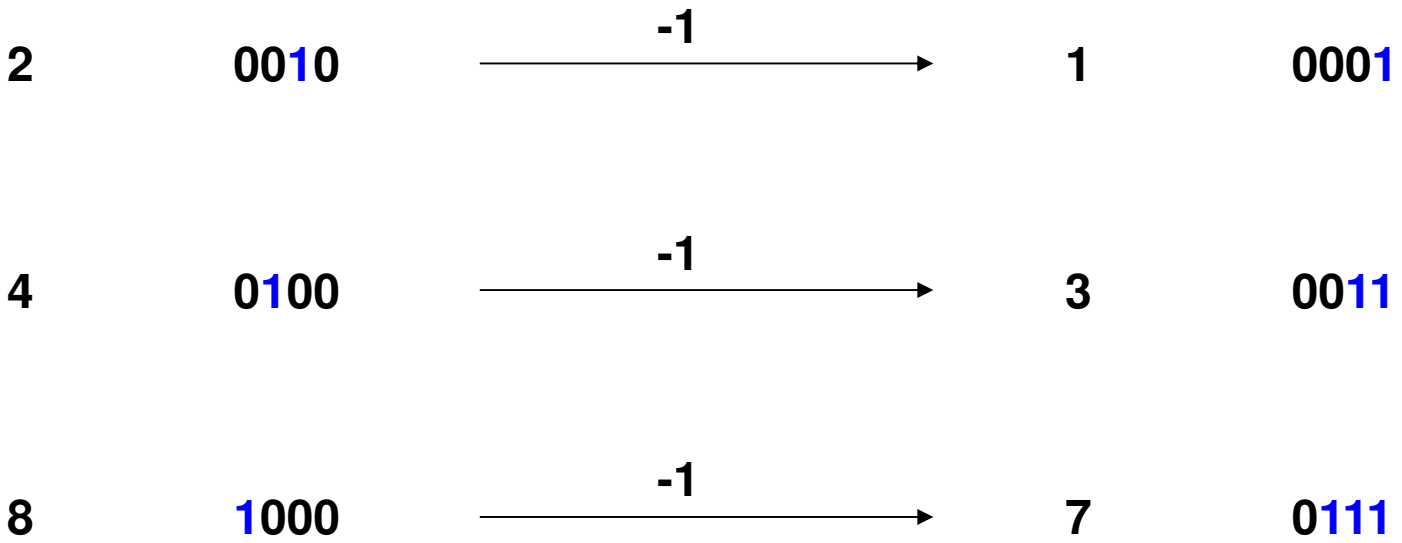
heap

```
1   int parent(int i)
2   {
3       return floor((i-1)/2);    // floor(i/2)
4   }
5
6   int getLeft(int i)
7   {
8       return (i*2) + 1;         // i*2
9   }
10
11  int getRight(int i)
12  {
13      return (i*2)+2;           // (i*2)+1
14  }
15
16
17
18
```

stampa



stampa



8	1000	
7	0111	&
0	0000	

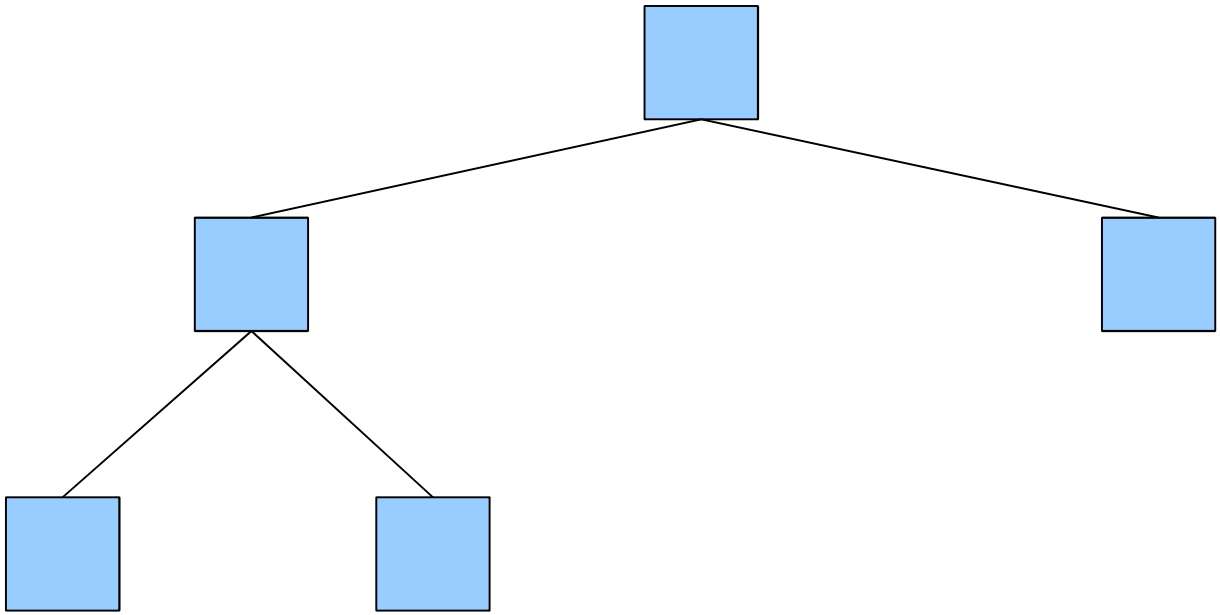
Print

```
1   bool isFirstChild( int i )
2   {
3       if( ( i!=0 ) && ( (i&(i-1)) == 0) )
4           return true;
5       else
6           return false;
7   }
8
9
10
11
12
13
14
15
16
17
18
```

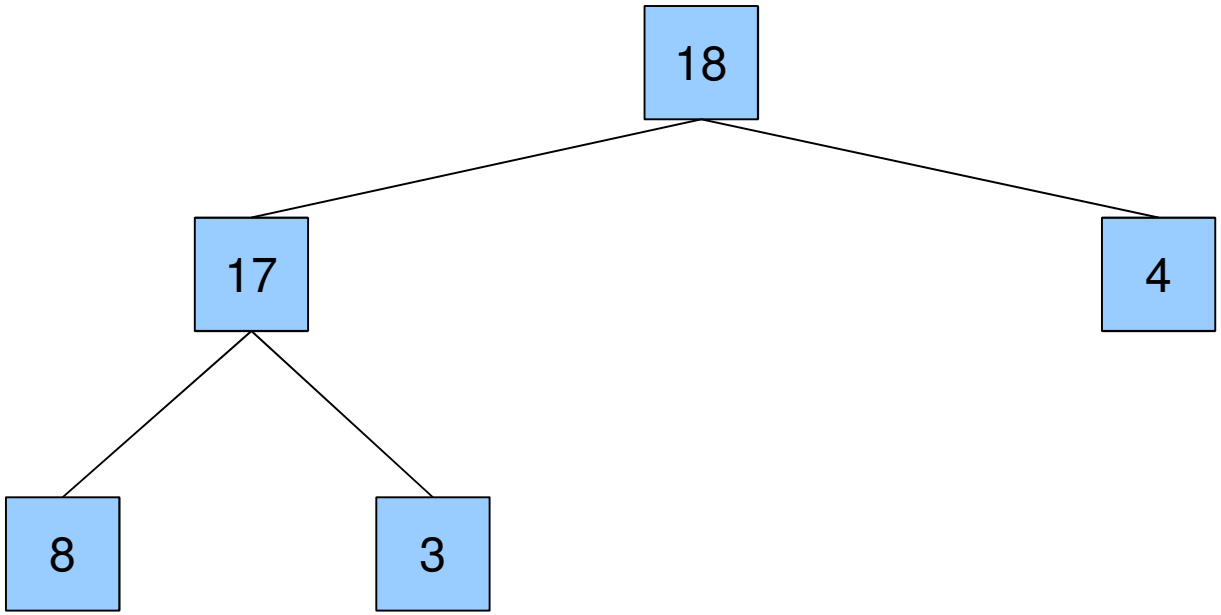
Print

```
1   bool isFirstChild( int i )
2   {
3       if( ( i!=0 ) && ( (i&(i-1)) == 0) )
4           return true;
5       else
6           return false;
7   }
8
9   void print()
10  {
11      for( int i=0 ; i < length_ ; ++i )
12      {
13          if( isFirstChild(i+1) )
14              cout << endl;
15          cout << data_[i] << "\t";
16      }
17      cout << endl;
18  }
```

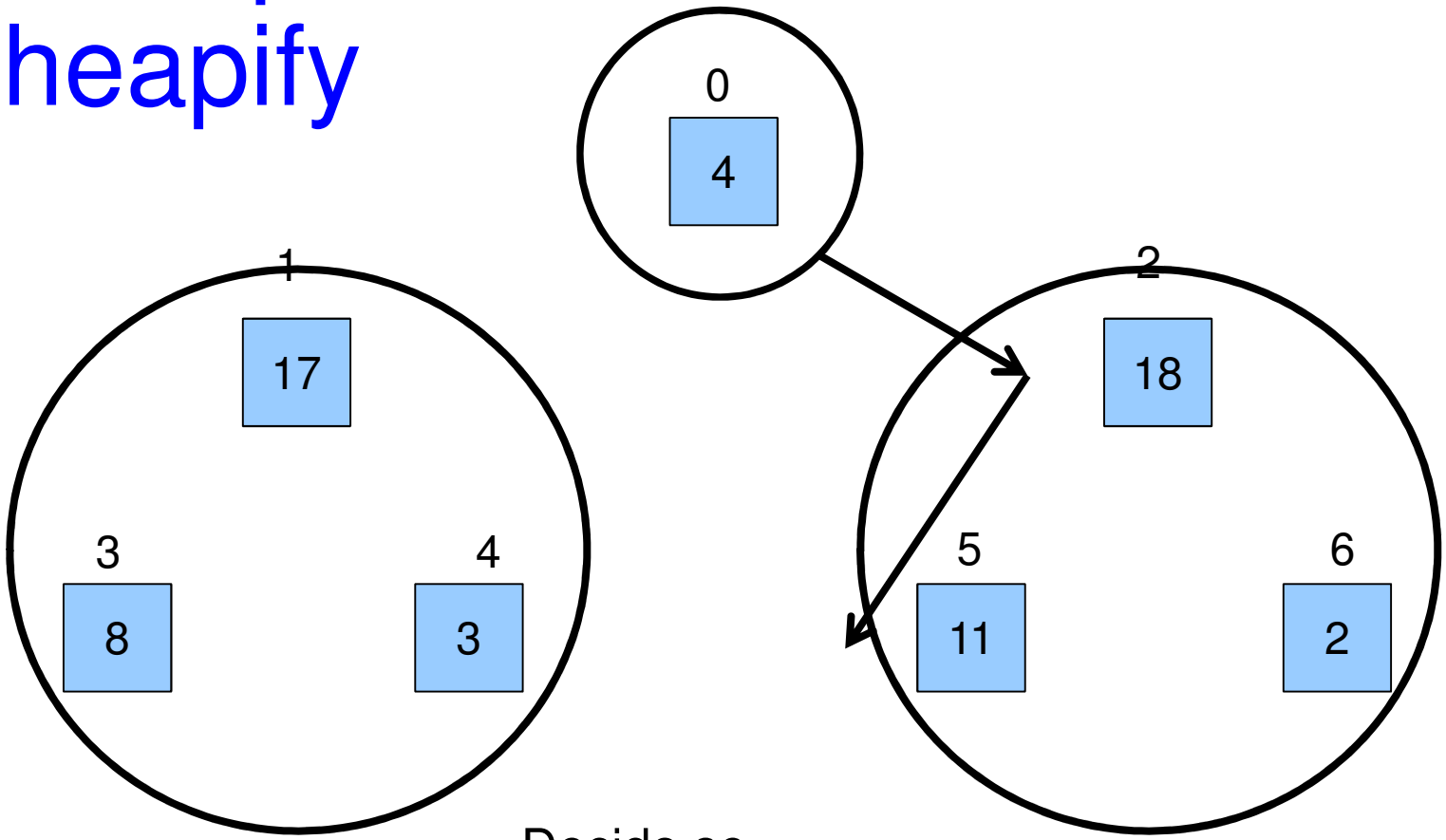
Heap Property



Heap Property



Esempio heapify



- Decido se
  - già ok?
  - andare a destra
  - andare a sinistra

heapify

```
1   void maxHeapify(int i)
2   {
3       // ottengo left e right
4
5
6
7       // (se ho figlio left) AND (left > i)
8           // left é più grande
9       // altrimenti
10          // i é più grande
11
12      // (se ho figlio right) AND (right > largest)
13          // right è più grande
14
15      // se i viola la proprietà di max-heap
16      {
17          // scambio i e il più grande
18          // controllo se l'albero che ho cambiato va bene
19      }
20  }
```

heapify

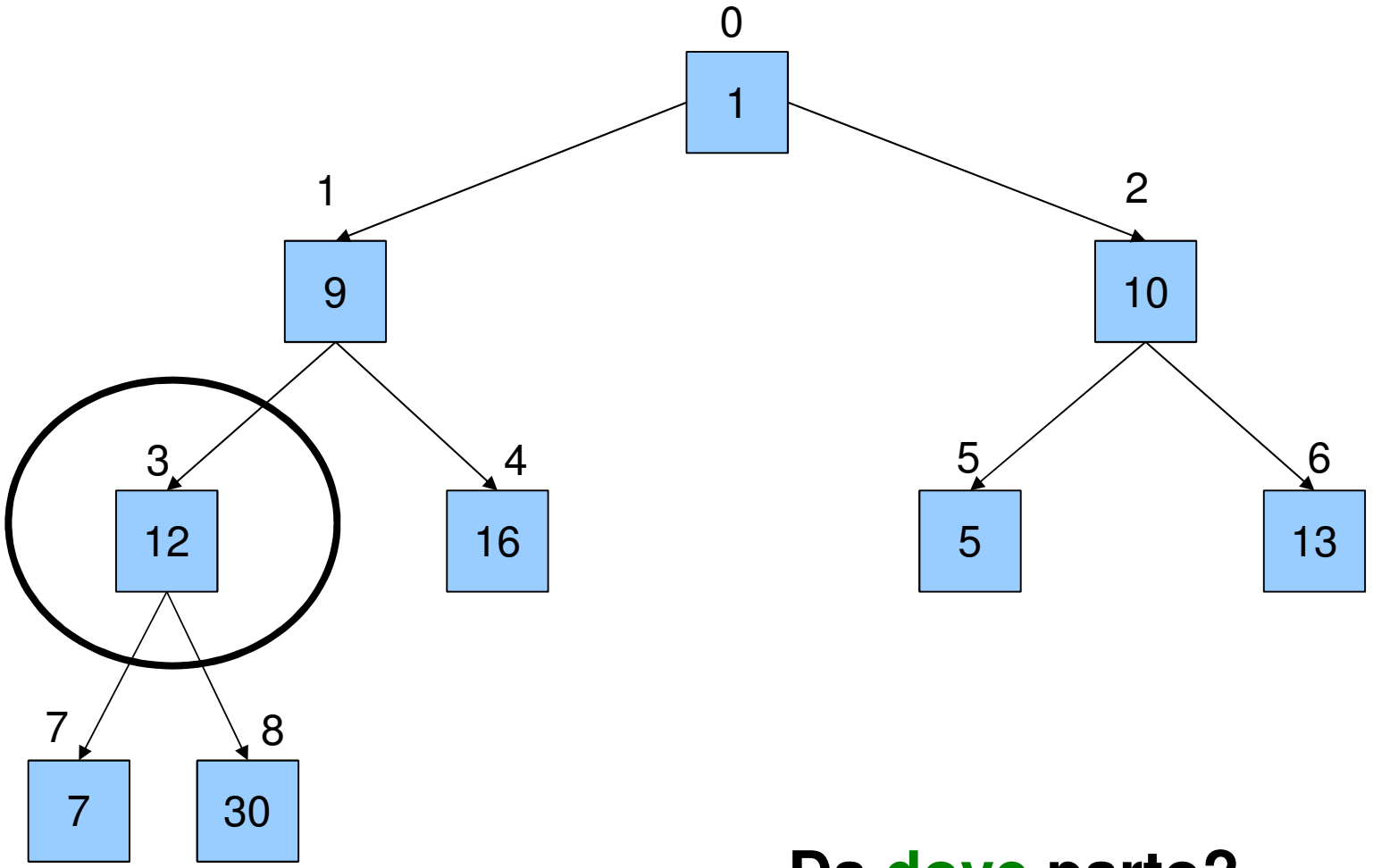
```
1   void maxHeapify(int i)
2   {
3       int left = getLeft(i);
4       int right = getRight(i);
5       int largest;
6
7       if((left < size_)&&(data_[left] > data_[i]))
8           largest = left;
9       else
10          largest = i;
11
12      if((right < size_)&&(data_[right] > data_[largest]))
13          largest = right;
14
15      if( largest != i )
16      {
17          scambia(i,largest);
18          maxHeapify(largest);
19      }
20  }
```

inizializzazione

Identifico + grande

Aggiorno albero

Build Heap

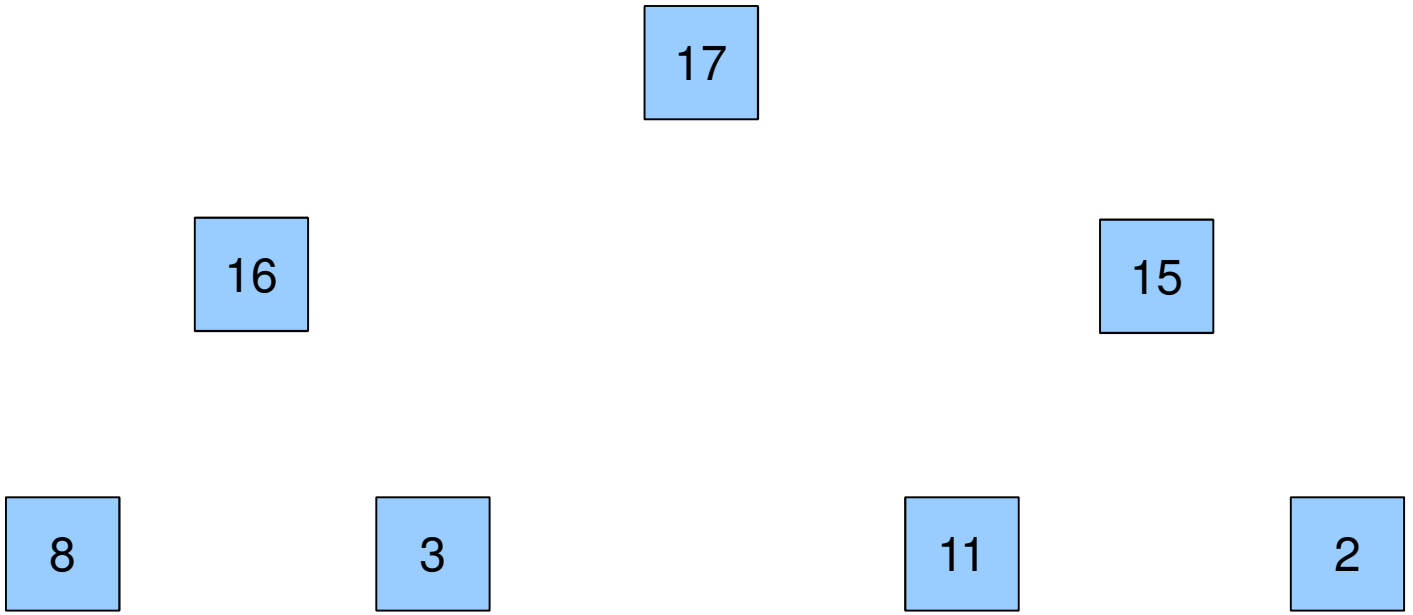


Da dove parto?

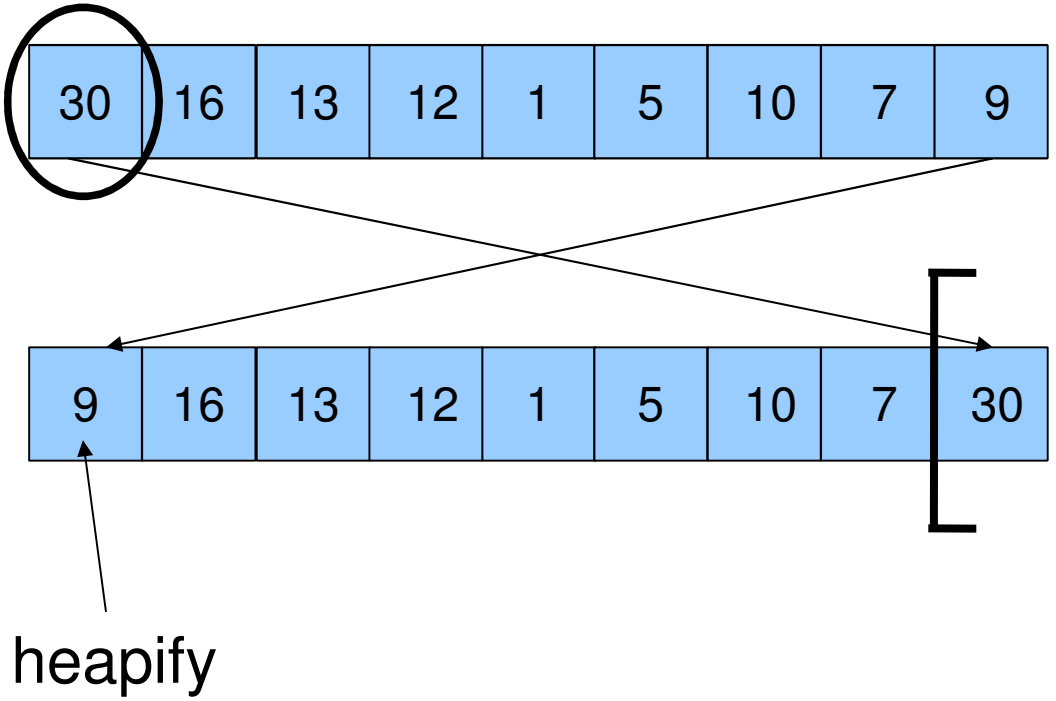
Build Heap

```
1   void buildMaxHeap()
2   {
3       size_ = length_;
4
5       int i = floor(length_/2)-1
6
7       for( ; i>=0 ; --i )
8       {
9           maxHeapify(i);
10          print();
11      }
12  }
13
14
15
16
17
18
19
20
```

Utilizzo

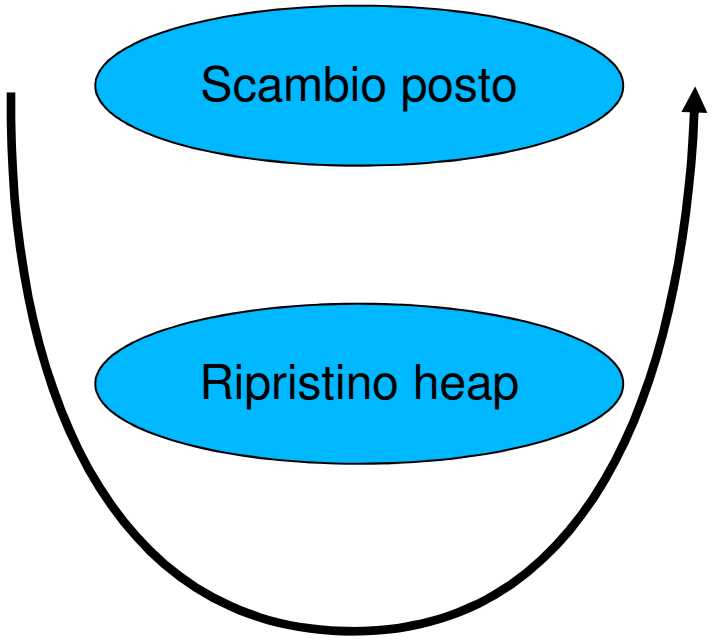


Esempio heapsort



heapsort

```
1   void heapSort()
2   {
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19  }
20
```



heapsort

```
1   void heapSort()
2   {
3
4       int i = length_-1
5
6       for( ; i>0 ; --i)
7       {
8
9
10          scambia(0,i);
11
12
13
14          --size_;
15
16          maxHeapify(0);
17      }
18
19  }
20
```

Programma completo

```
1   int main()
2   {
3       Heap hp;
4
5       hp.fill();
6       hp.print();
7
8
9       hp.buildMaxHeap();
10      hp.print();
11
12      hp.heapSort();
13      hp.printArray();
14
15      return 0;
16
17  }
18
```

Esercizi (per casa)

- Aggiunta nodo
- Eliminazione nodo
- Aumento Valore

Heap STL

```
#include <algorithm>
make_heap( inizio , fine )
pop_heap( inizio , fine )

#include <queue>
priority_queue<int> prioQ
prioQ.push(val)
prioQ.top()
prioQ.pop()
```

Algorithms

```
1   #include <vector>
2   #include <algorithm>
3
4
5   vector<int> vect;
6
7   for( int i = 0 ; i<quanti ; ++i )
8   {
9       cin >> val;
10      vect.push_back(val);
11  }
12
13  make_heap(vect.begin(),vect.end());
14
15  while(!vect.empty())
16  {
17      cout << "top " << *vect.begin() << endl;
18      pop_heap(vect.begin(),vect.end());
19      vect.pop_back();
20  }
```

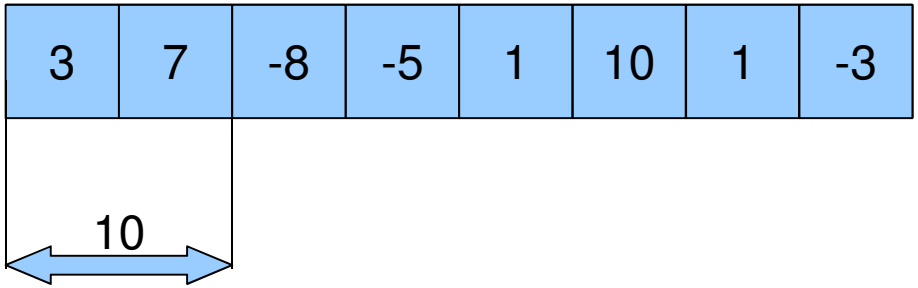
priority_queue

```
1   #include <queue>      // std::priority_queue
2
3   priority_queue<int> prioQ;
4
5   for( int i = 0 ; i<quanti ; ++i )
6   {
7       cin >> val;
8       prioQ.push(val);
9   }
10
11  while(!prioQ.empty())
12  {
13      cout << "top " << prioQ.top() << endl;
14      prioQ.pop();
15  }
16
17
18
19
20
```

Esercizi

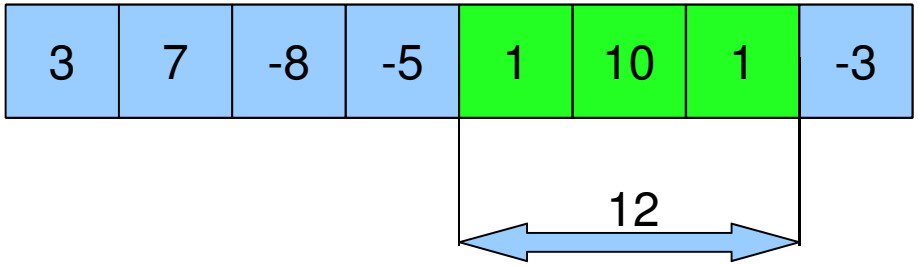
- Esperimenti
  - Utilizzo Heap fatto a mano
  - Heapsort VS MergeSort
  - Priority_queue

Somma Massima



- Input: array
- Output: somma massima

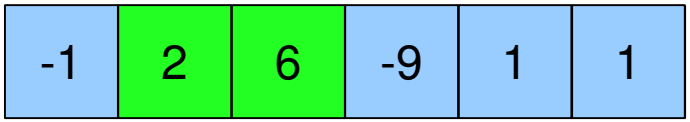
- Esempio



proprietà

la somma degli elementi del sotto array di somma massima è sempre positiva

Il valore precedente al primo valore del sotto array di somma massima è negativo

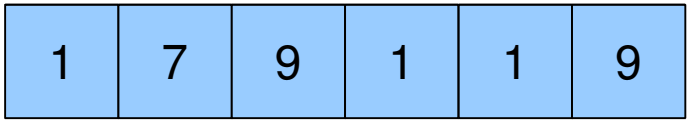


Soluzione 3

```
1   int somme3(int a[] , int size )
2   {
3       int somma;
4       int i;
5       int max=a[0];
6       somma = 0;
7       for(i=0; i<size; i++)
8       {
9           if(somma > 0) somma+=a[i];
10          else somma=a[i];
11
12          if(somma > max) max=somma;
13      }
14      return max;
15  }
```

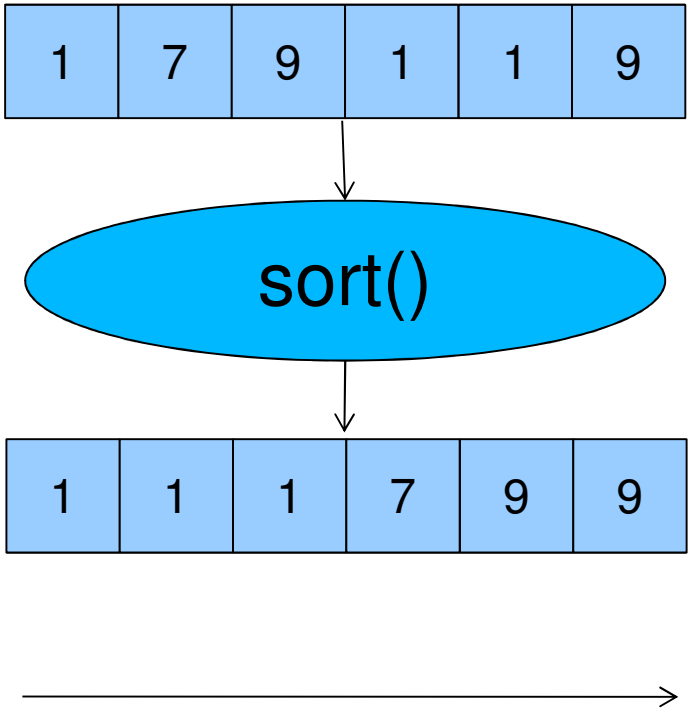
$\Theta(n)$

Distinti in Array

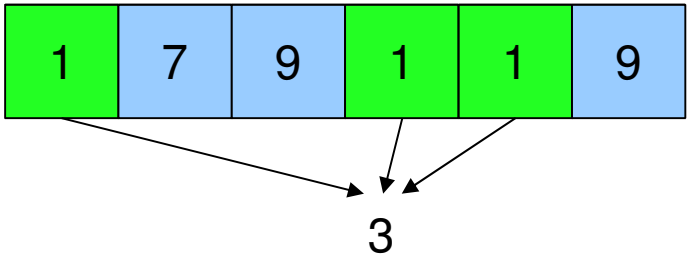


- Input: elementi array
- Output: array senza duplicati

Distinti in Array (2)

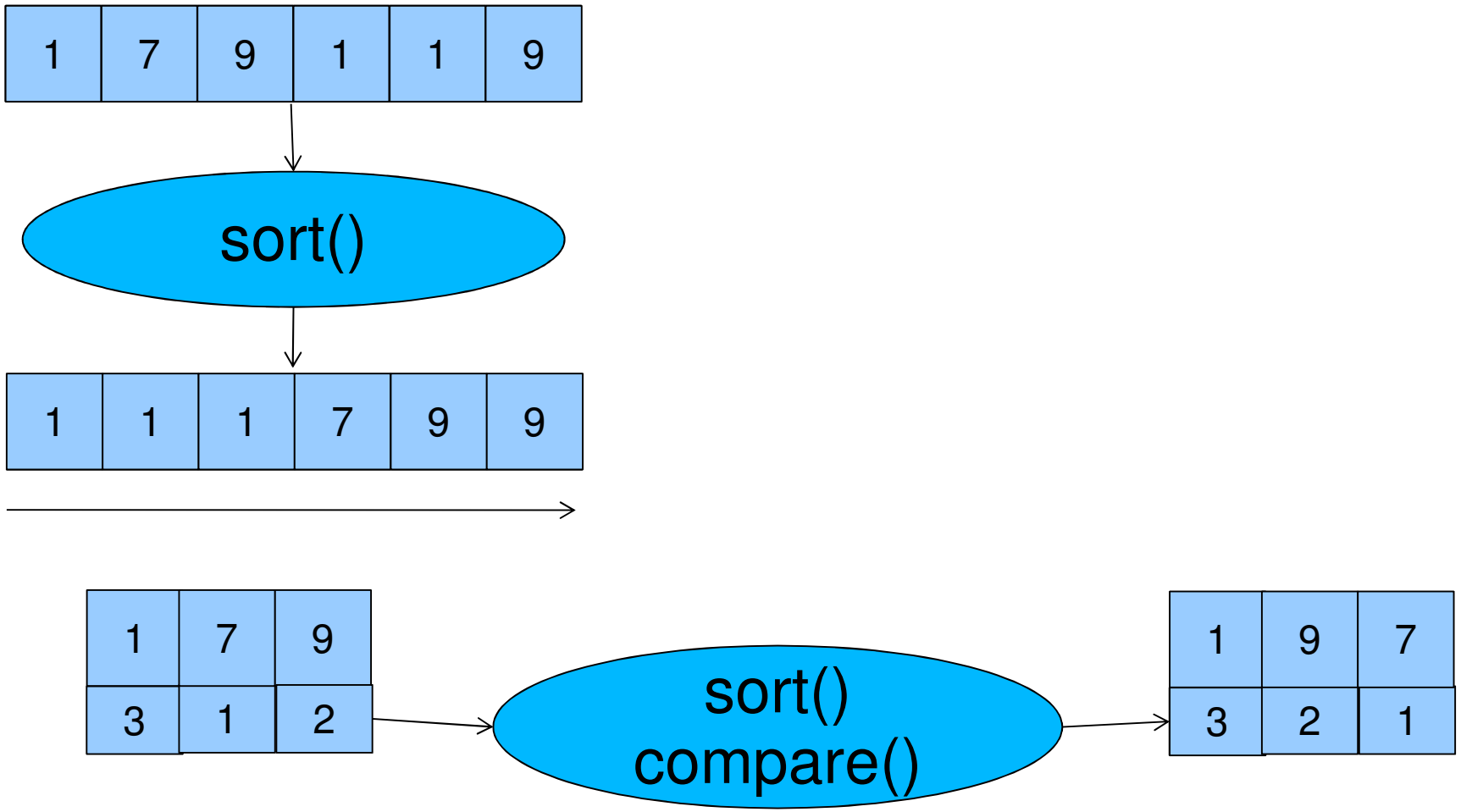


K interi più frequenti



- Input: elementi array , intero k
- Output: primi k valori più frequenti

K interi più frequenti



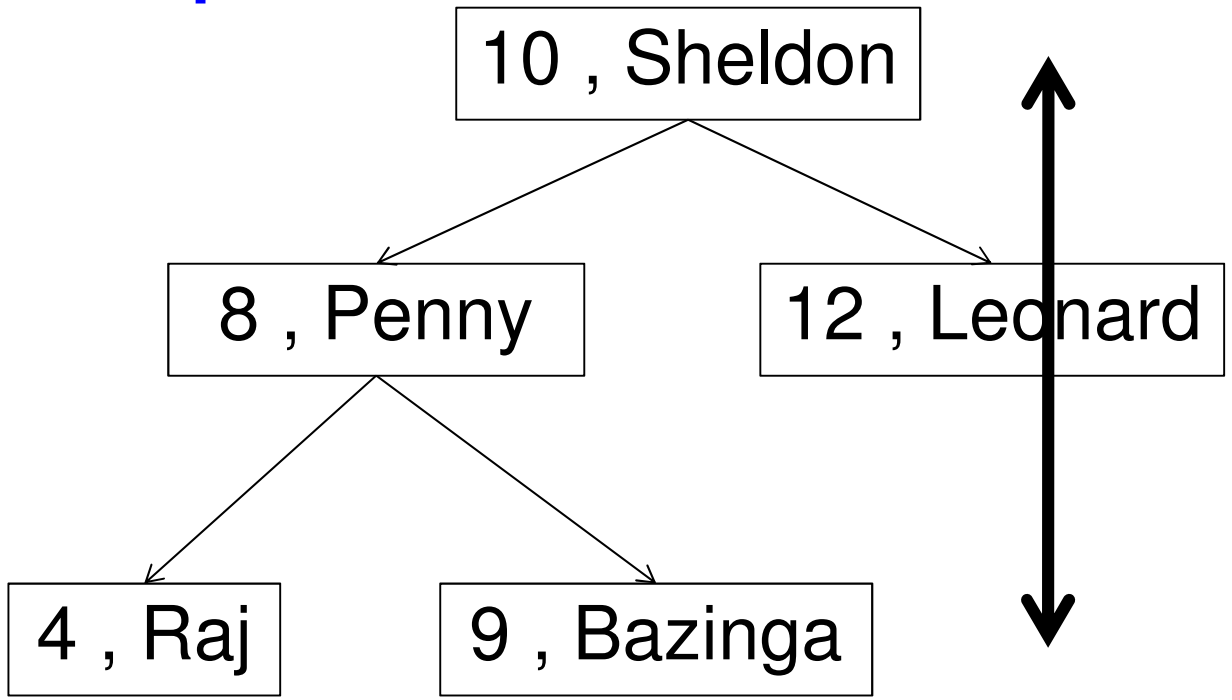
Albero Binario a etichette complesse

- Input:
  - Un intero **N**
  - Un intero **H**
  - N coppie **[intero,stringa]**
- Operazioni:
  - Inserire le N coppie in un albero binario di ricerca (usando il valore intero come chiave)
- Output:
  - stringhe che si trovano in nodi ad altezza H, stampate in ordine lessicografico

Albero Binario a etichette complesse

input

```
5
2
10
Sheldon
12
Leonard
8
Penny
4
Raj
9
Bazinga
```



output

```
Bazinga
Raj
```

Analisi

- Input:
  - Un intero N
  - Un intero H
  - N coppie [intero,stringa]
- Operazioni:
  - Inserire le N coppie in un albero binario di ricerca
- Output:
  - stringhe che si trovano in nodi ad altezza H, stampate in ordine lessicografico

Analisi

Implementare struttura dati che supporti
- Albero binario
- Etichette multi valore

```
struct node
{
        int key;
        string str;
        struct node* right;
        struct node* left;
} Node;
```

Funzioni

- Insert su albero binario → insert()
- Trovare nodi ad altezza H → visita+altezza
- Sort su string → sort+compare

Trova nodi ad altezza H

```
1   void getStringList(    Node* node,
2                          int curr_h ,
3                          int H,
4                          vector<string> & strList )
5   {
6       if(node==NULL) return;
7
8       if(curr_h==H)
9       {
10          strList.push_back(node->str);
11          return;
12      }
13
14      getStringList(node->left,curr_h+1,H,strList);
15      getStringList(node->right,curr_h+1,H,strList);
16
17      return;
18  }
19
20
```

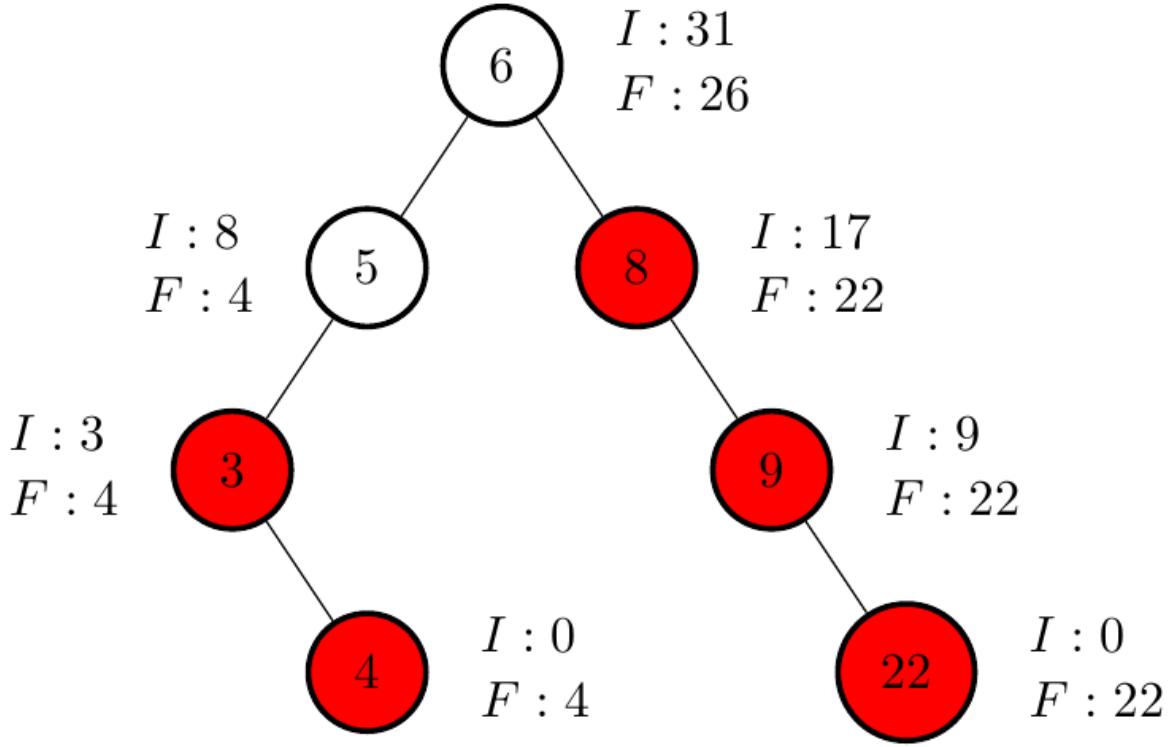
Somma Nodi

- Input:
  - Un intero N
  - N interi
- Operazioni:
  - Inserire gli N interi in un albero binario di ricerca
  - Per ogni nodo *u*, calcolare *I(u)* e *F(u)*
- Output:
  - Stampare le etichette dei nodi tali che *I(u)* <= *F(u)*

Somma Nodi (2)

I(u)*: somma delle chiavi dei nodi interni del sottoalbero radicato in *u

F(u)*: somma delle chiavi delle foglie del sottoalbero radicato in *u



Calcolo I(u) e F(u)

- Devo visitare tutto l'albero.
- I valori di I(u) e F(u) di un nodo padre, dipendono dagli stessi valori calcolati per i nodi figli.
- Di quali nodi posso calcolare I(u) e F(u) "al volo"?

- **Suggerimento:** come facevamo a calcolare l'altezza di un nodo? (relazione padre/figli)