Algoritmi e Strutture Dati

Lezione 4

http://mlpi.ing.unipi.it/alfeo

Antonio Luca alfeo

luca.alfeo@ing.unipi.it

Sommario

- Esercizi ultimo laboratorio
- Heap
- Ordinamento tramite Heap
- Hashing
- Hashing e tipi di input
- Esercizi

Esercizio: nodi “concordi” e “discordi”

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca (ABR). Si definiscono:

- Concordi i nodi c caratterizzati da altezza del nodo pari (dispari) e etichetta pari (dispari).
- Discordi i nodi d con altezza pari e etichetta dispari, o viceversa.

L’ altezza del nodo x corrisponde al numero di nodi compresi tra x e la radice dell’albero, x escluso. La radice ha altezza 0. Scrivere un programma che:

- legga da tastiera N, il numero di interi da memorizzare nell’albero;
- legga da tastiera N interi, ovvero le etichette dei nodi da inserire nell’ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette ≤ vanno inserite a sinistra);
- stampi la differenza tra la sommatoria S delle etichette dei nodi discordi e la stessa dei nodi concordi

$$S = \sum label(d) - \sum label(c)$$

Possibile Soluzione: nodi “concordi” e “discordi”

```
#include <iostream>      // std::cout
#include <algorithm>     // std::sort
#include <vector>        // std::vector
#include <fstream>
#include <math.h>        /* floor */
#include <stdlib.h>
#include <cmath>         /* pow  */

using namespace std;


struct Node
{
    int value;
    Node * left;
    Node * right;

    Node( int i ): value(i) , left(NULL) , right(NULL) {}
};
```

```
class BinTree
{
    Node * root_;
public:
    BinTree() { root_ = NULL ; }

    Node * getRoot() { return root_; cout << "getRoot" << endl;}

    void insert( int i )
    {
        Node * node = new Node(i);

        Node * pre = NULL;
        Node * post = root_;
        while( post != NULL)
        {
            pre = post;
            if( i <= post->value )
            {
                post = post->left;
            }
            else
            {
                post = post->right;
            }
        }

        if( pre == NULL )
            root_ = node;
        else if( i <= pre->value )
        {
            pre->left = node;
        }
        else
        {
            pre->right = node;
        }
        return;
    }
};
```

```
int layerTree( Node * tree, int altezza)
{
     // Nodo non trovato
     if( tree == NULL)  {
        return 0;
     }

     if (altezza++%2 != tree->value%2)  {
          //cout<<tree->value<<" : "<<altezza-1<<" discorde"<<endl;
          return layerTree( tree->left , altezza) + layerTree( tree->right , altezza) + tree->value;
     }
     else   {
          //cout<<tree->value<<" : "<<altezza-1<<" concorde"<<endl;
          return layerTree( tree->left , altezza) + layerTree( tree->right , altezza) - tree->value;
     }
}


int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )   {
        cin >> x;
        albero.insert(x);
    }

    cout<<layerTree(albero.getRoot(), 0)<<endl;
}
```

Esercizio: nodi “completi”

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca (ABR). Si definiscono completi i nodi c aventi due nodi figli, incompleti i nodi i restanti. Scrivere un programma che:

- legga da tastiera N, il numero di interi da memorizzare nell’albero;
- legga da tastiera N interi, ovvero le etichette dei nodi da inserire nell’ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette ≤ vanno inserite a sinistra);
- stampi la differenza tra la sommatoria S delle etichette dei nodi completi e la stessa dei nodi incompleti

$$S = \sum label(c) - \sum label(i)$$

Possibile soluzione: nodi “completi”

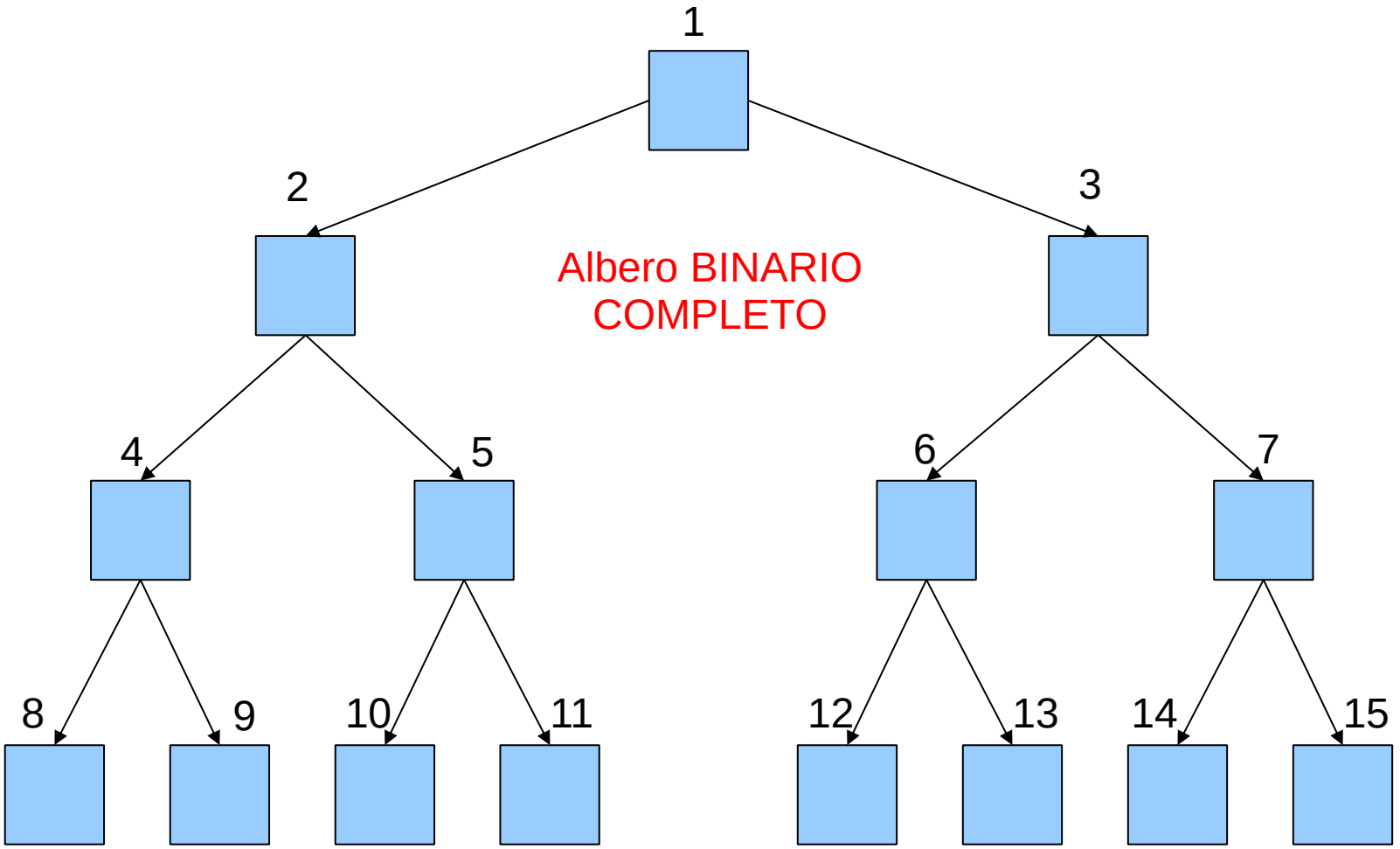
```
#include <iostream>      // std::cout
#include <algorithm>     // std::sort
#include <vector>        // std::vector
#include <fstream>
#include <math.h>        /* floor */
#include <stdlib.h>
#include <cmath>         /* pow  */

using namespace std;


struct Node
{
    int value;
    Node * left;
    Node * right;

    Node( int i ): value(i) , left(NULL) , right(NULL) {}
};
```

```
class BinTree
{
    Node * root_;
public:
    BinTree() { root_ = NULL ; }

    Node * getRoot() { return root_; cout << "getRoot" << endl;}

    void insert( int i )
    {
        Node * node = new Node(i);

        Node * pre = NULL;
        Node * post = root_;
        while( post != NULL)
        {
            pre = post;
            if( i <= post->value )
            {
                post = post->left;
            }
            else
            {
                post = post->right;
            }
        }

        if( pre == NULL )
            root_ = node;
        else if( i <= pre->value )
        {
            pre->left = node;
        }
        else
        {
            pre->right = node;
        }
        return;
    }
};
```

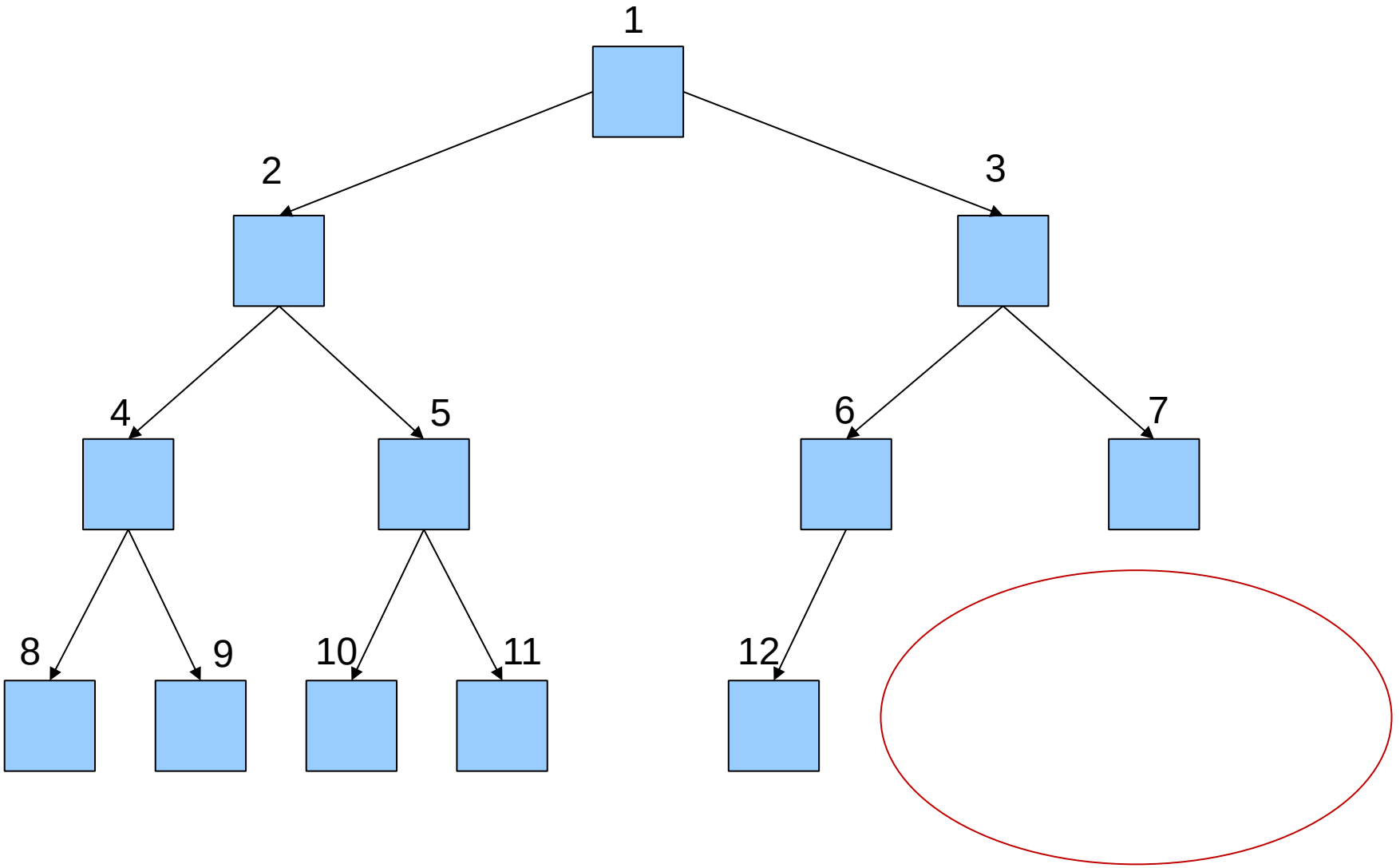
```
int complete( Node * tree)
{
     // Nodo non trovato
     if( tree == NULL)  {
        return 0;
     }

     if (tree->left != NULL && tree->right != NULL)  {
          return complete( tree->left ) + complete( tree->right ) + tree->value;
     }
     else   {
          return complete( tree->left ) + complete( tree->right ) - tree->value;
     }
}


int main()
{
     int N ;
     int x ;
     BinTree albero ;

     cin >> N ;

     // riempio l' albero
     for(int i=0 ; i<N ; ++i )   {
          cin >> x;
          albero.insert(x);
     }

     cout<<complete(albero.getRoot())<<endl;
}
```

HEAP

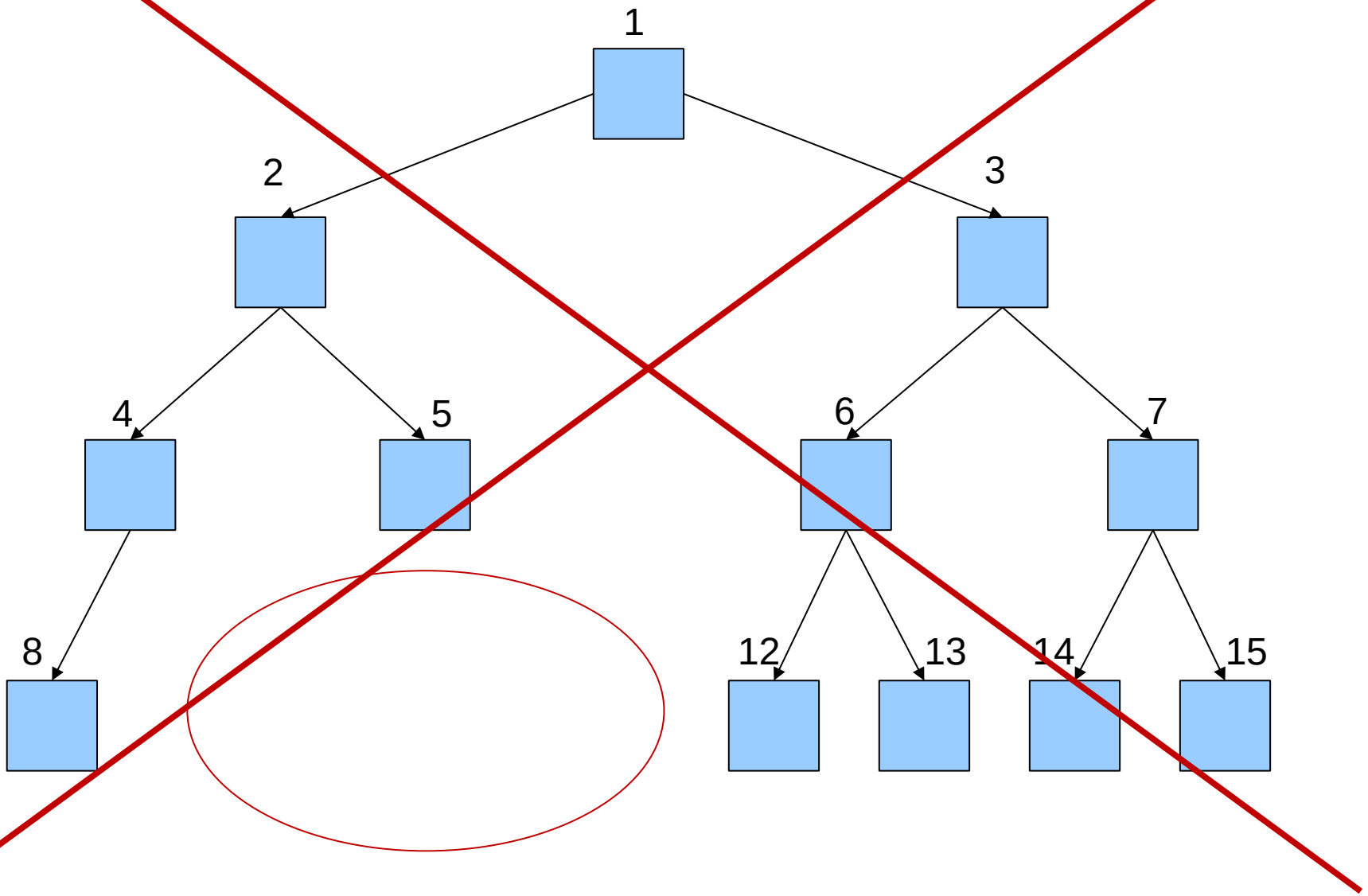
Proprieta' strutturali Heap



Heap, non completo

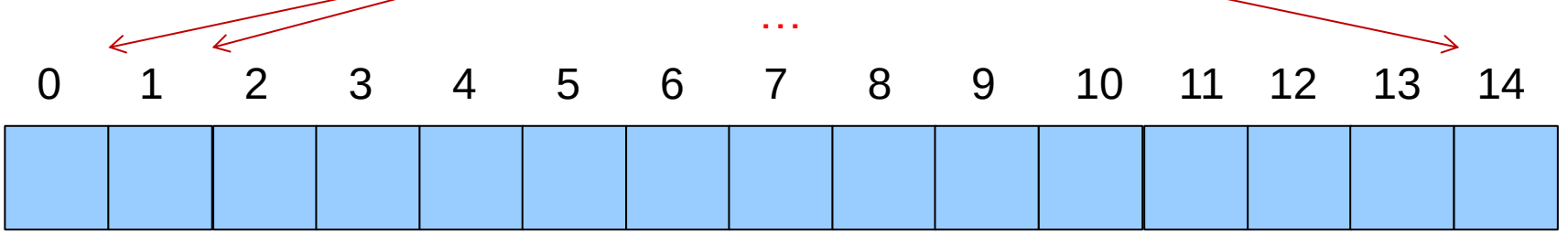


Heap, non completo

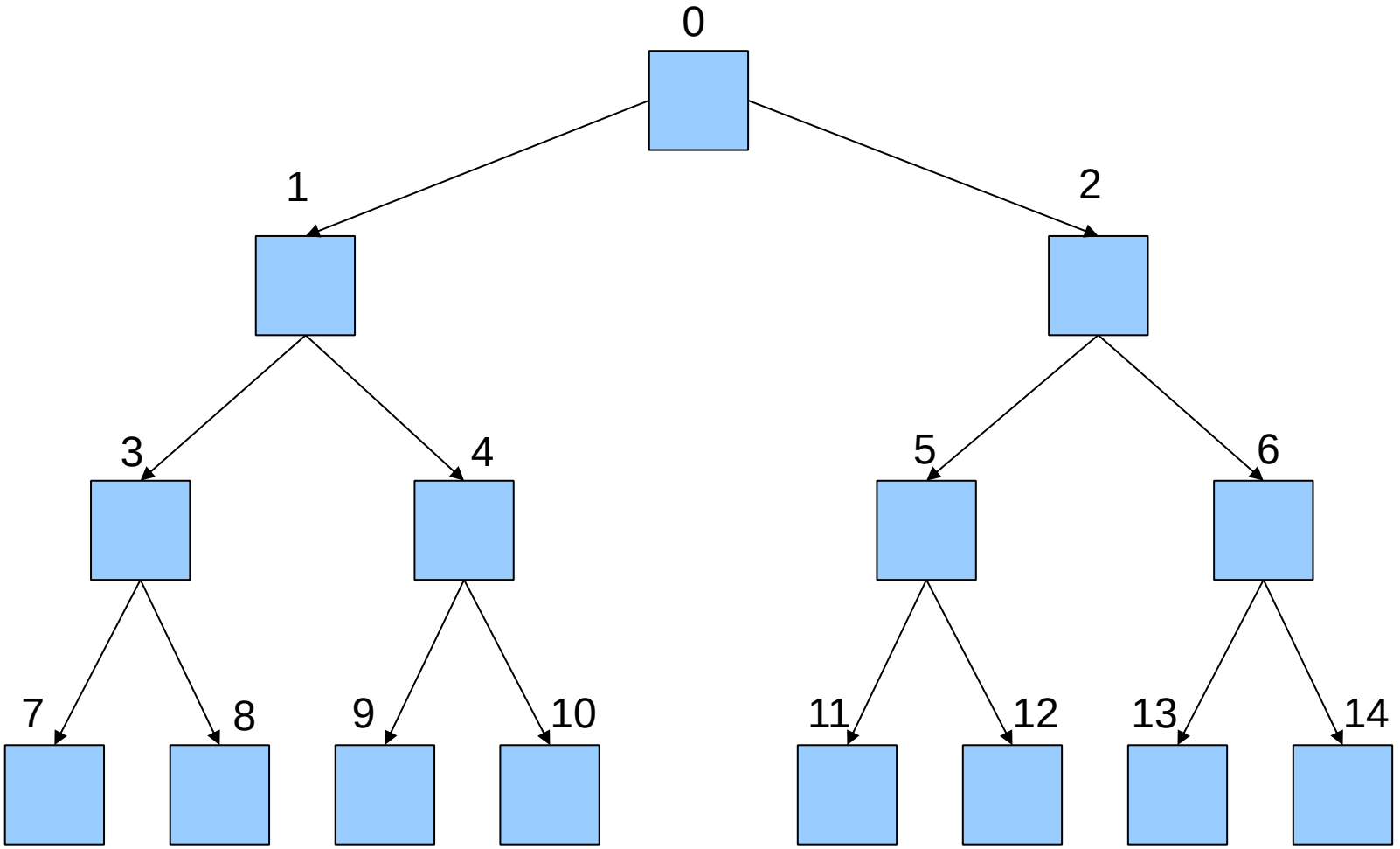


Heap <-> Array

Normalmente riferiamo la **posizione nello Heap**, ma se volessimo ragionare in termini di **indici delle posizioni nell'array?**



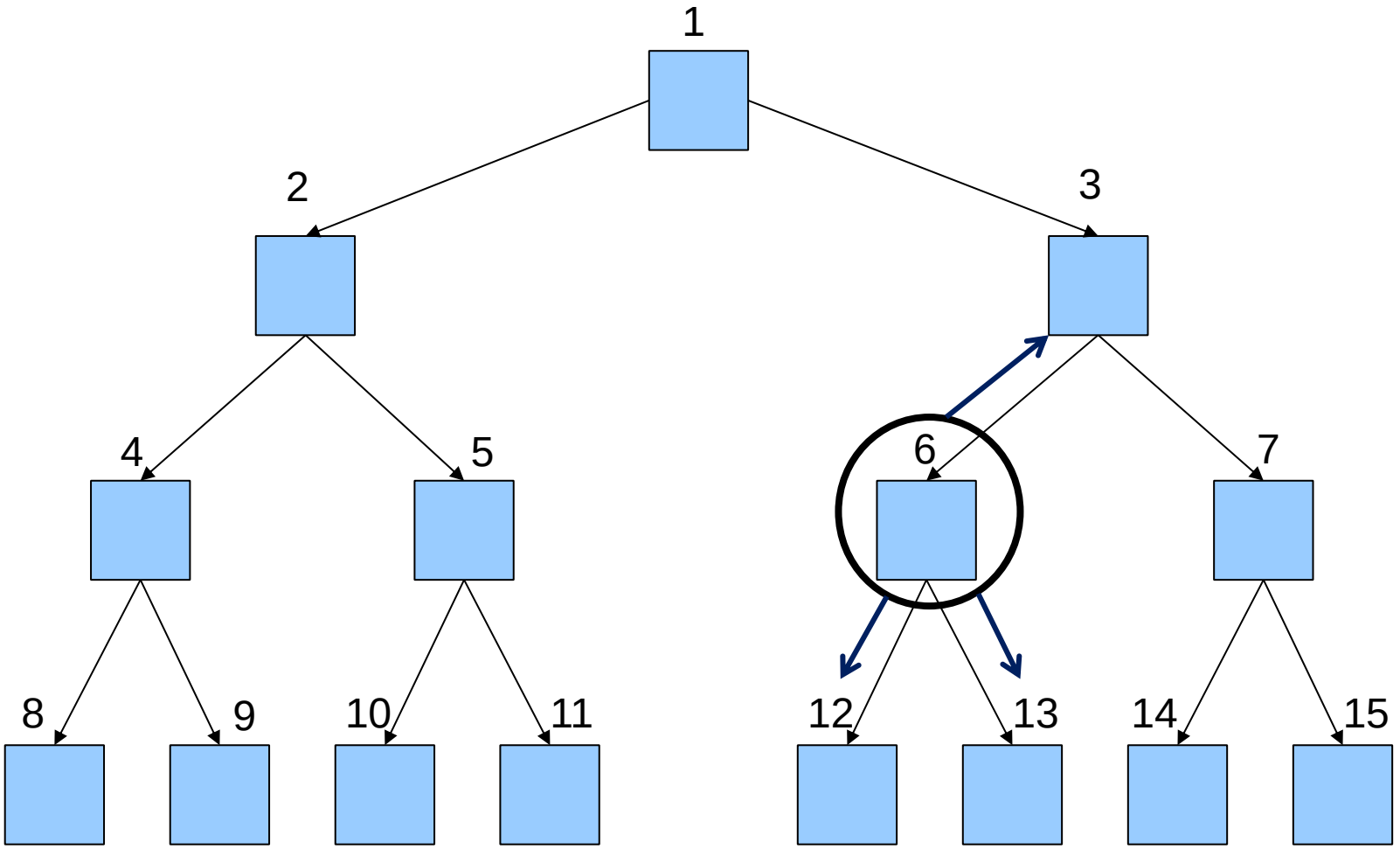
Heap <-> Array



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  }
```

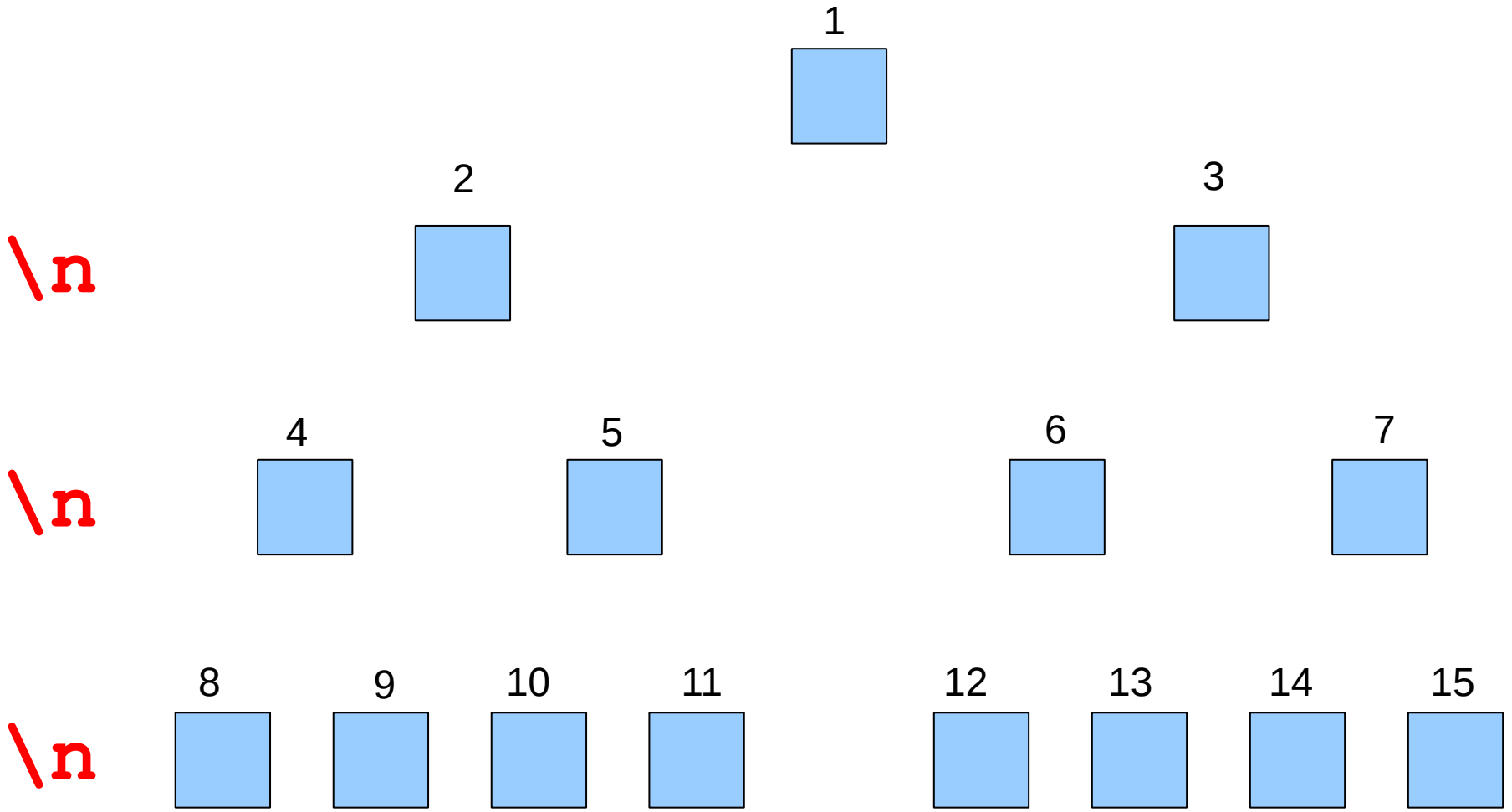
Navigare lo Heap grazie alle sue proprieta' strutturali



Navigare lo 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
```

Stampare il contenuto dello Heap



Quando andare a capo?

2	**0010**	**-1** →	**1**	**0001**
4	**0100**	**-1** →	**3**	**0011**
8	**1000**	**-1** →	**7**	**0111**

8	**1000**	
7	**0111**	**&**
0	**0000**	

Print

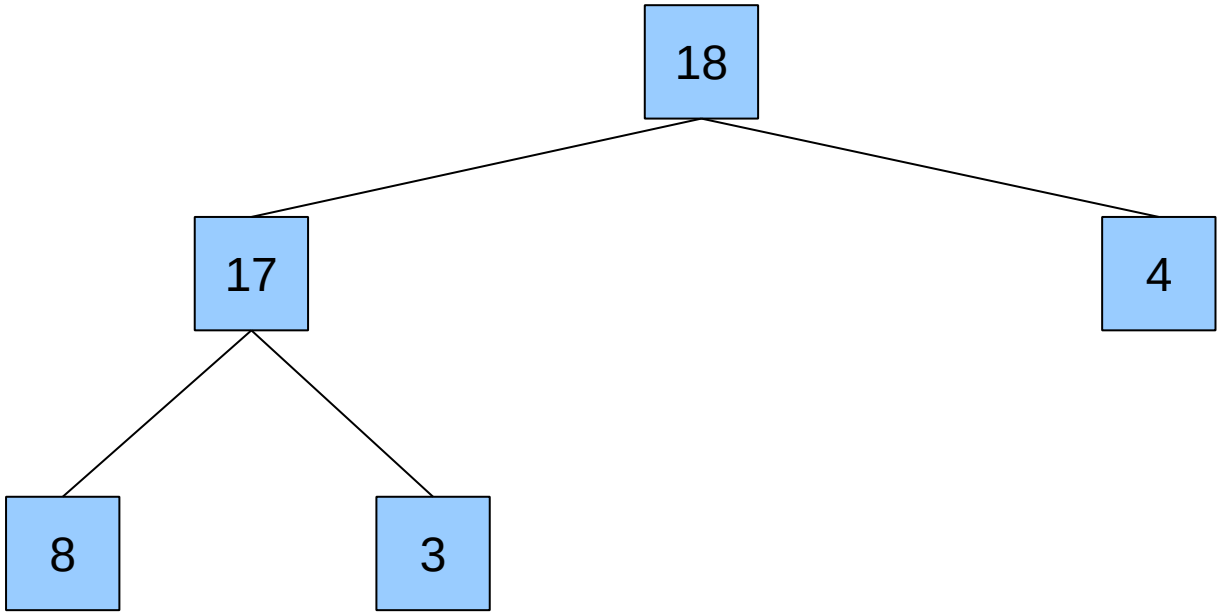
```
1   bool isFirstChild( int i )
2   {
3       if( ( i!=0 ) && ( (i&(i-1)) == 0) )
4           return true;
5       else
6           return false;
7   }
8
9
10
11
12
13
14
15
16
17
18
```

Quando andare a capo

Print

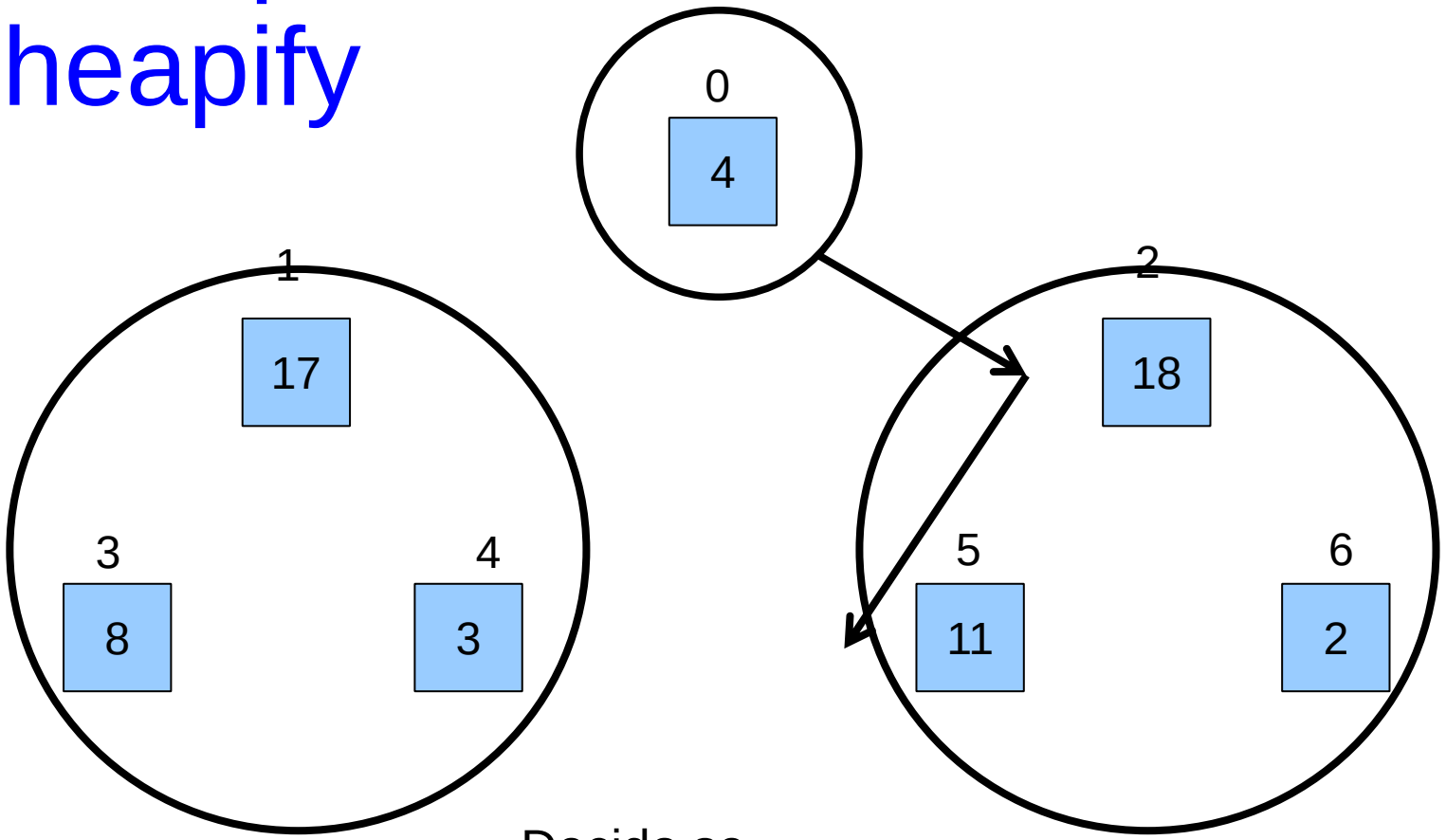
```
1   bool isFirstChild( int i )
2   {
3       if( ( i!=0 ) && ( (i&(i-1)) == 0) )
4           return true;
5       else
6           return false;
7   }
8
9   void print()
10  {
11      for( int i=0 ; i < length_ ; ++i )
12      {
13          if( isFirstChild(i+1) )
14              cout << endl;
15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

Proprieta' valori di un Heap



Consideriamo un max-heap:
ogni nodo padre deve avere etichetta > delle etichette dei suoi figli!

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

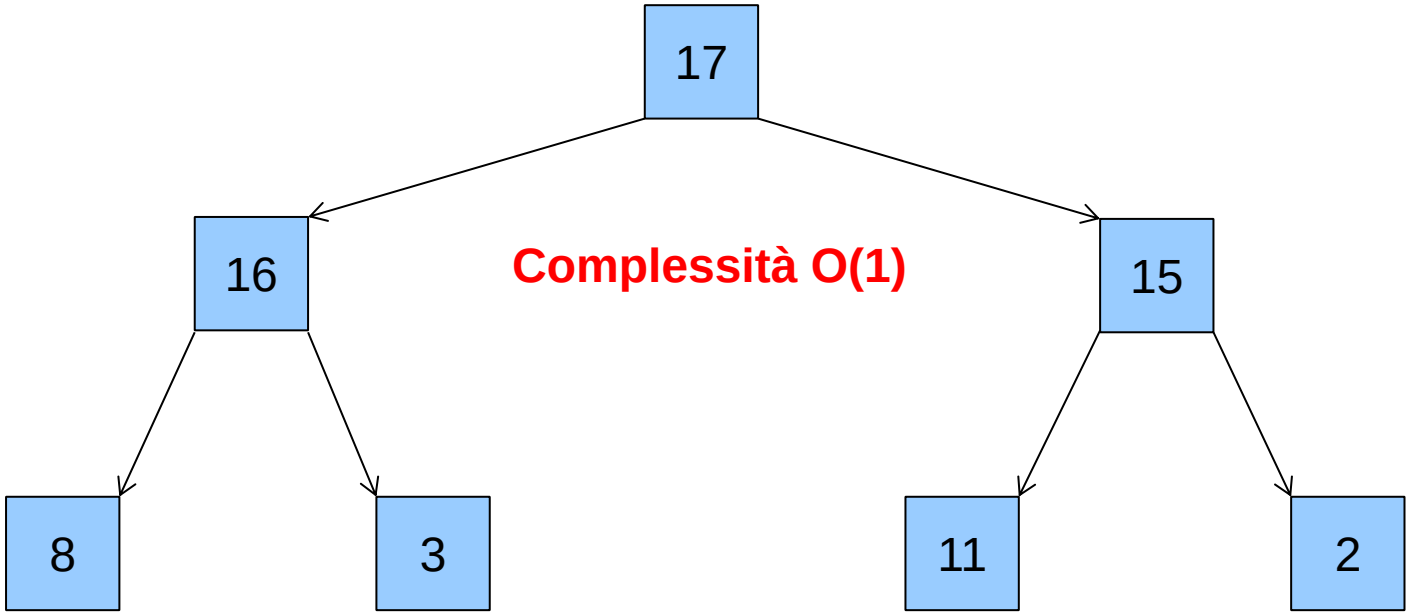
Identifico + grande

Aggiorno albero

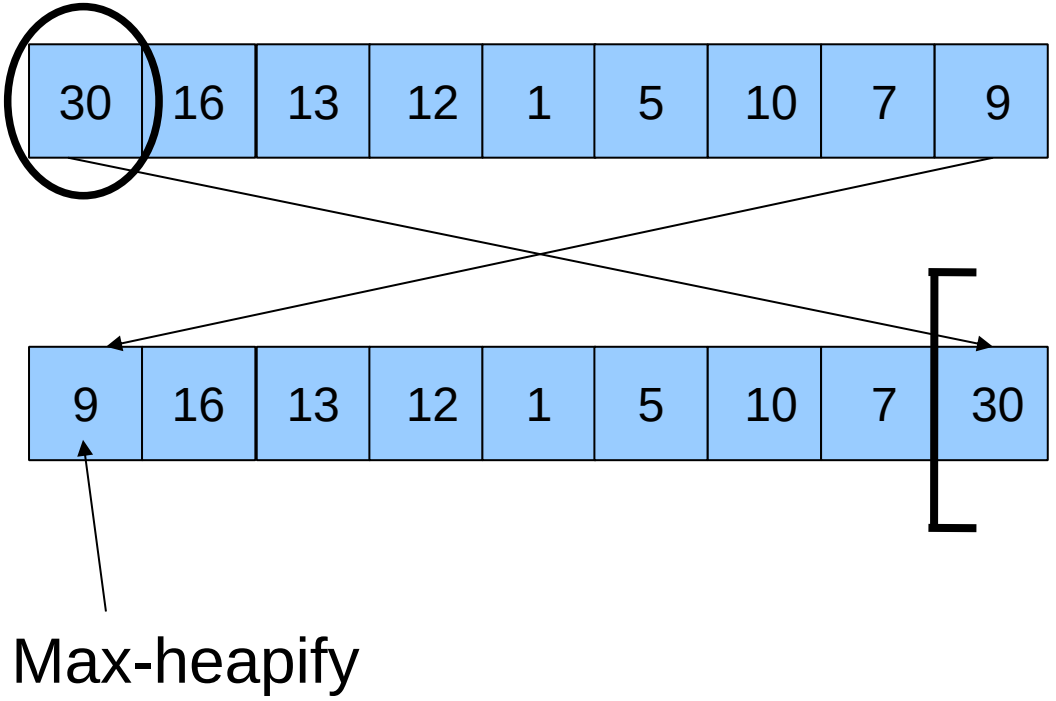
Build Heap

```
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
```

Trovare il massimo?



Heapsort



Ad ogni passo:

- Scambio l'elemento max (la radice) con l'ultimo elemento

- Riduco la dimensione dello heap (size)

- Applico la max-heapify sulla radice

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
```

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
```

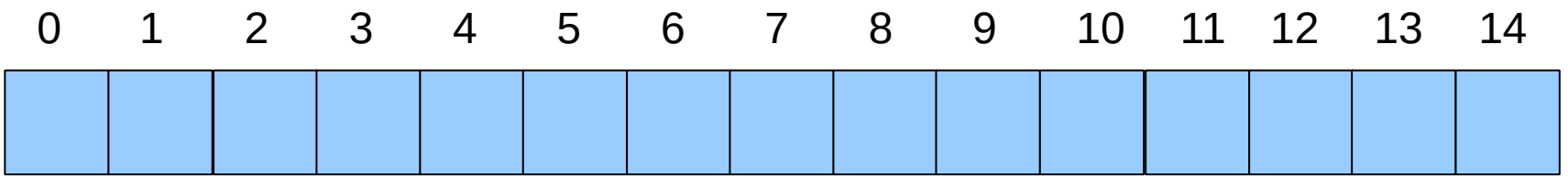
Esercizi

- Esercizi
  - Aggiunta nodo ad un heap
  - Eliminazione nodo da un heap
  - Aumento Valore di un nodo di un heap

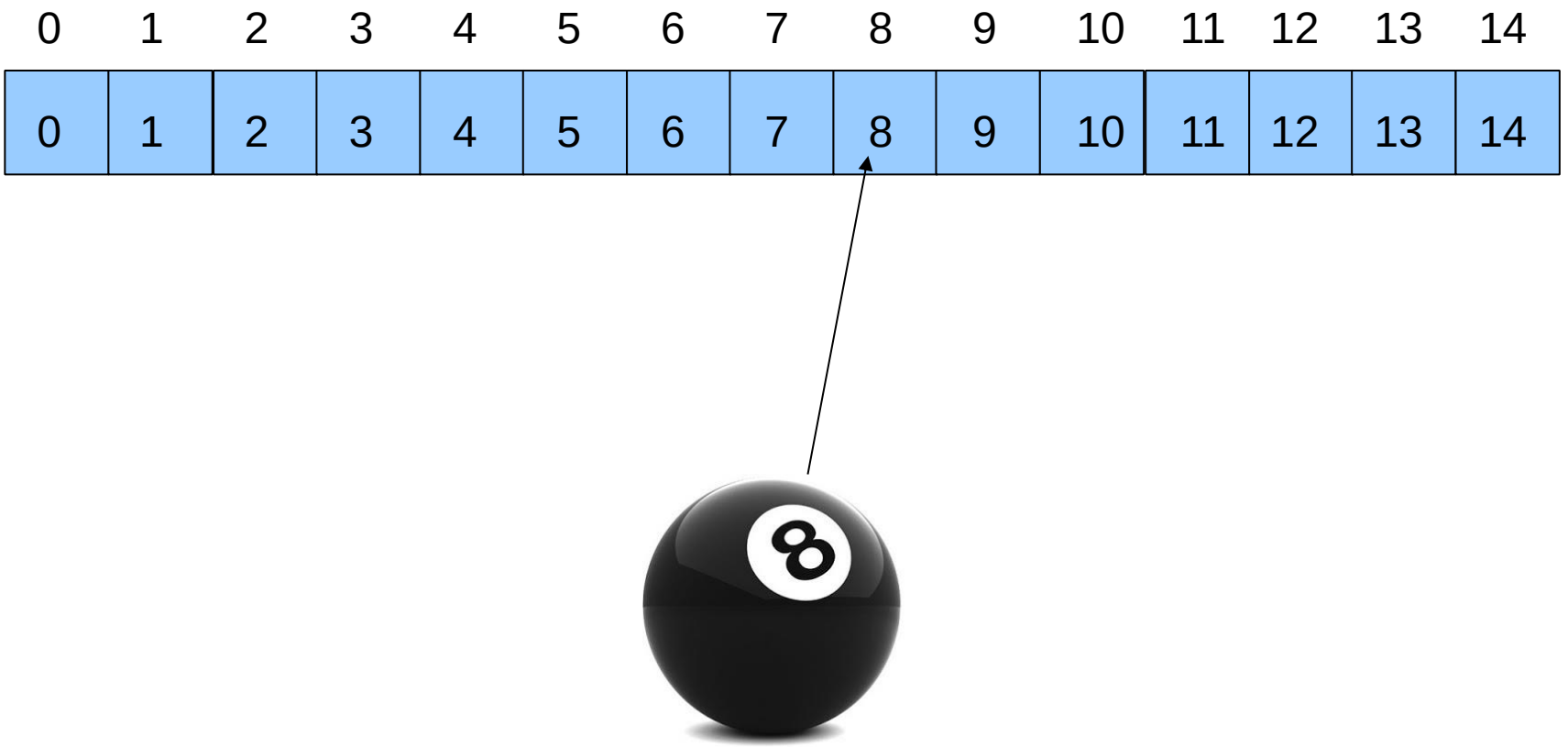
- Esperimenti
  - Utilizzo Heap fatto a mano
  - Heapsort VS MergeSort
  - Priority_queue

HASH

Array ad indirizzamento diretto

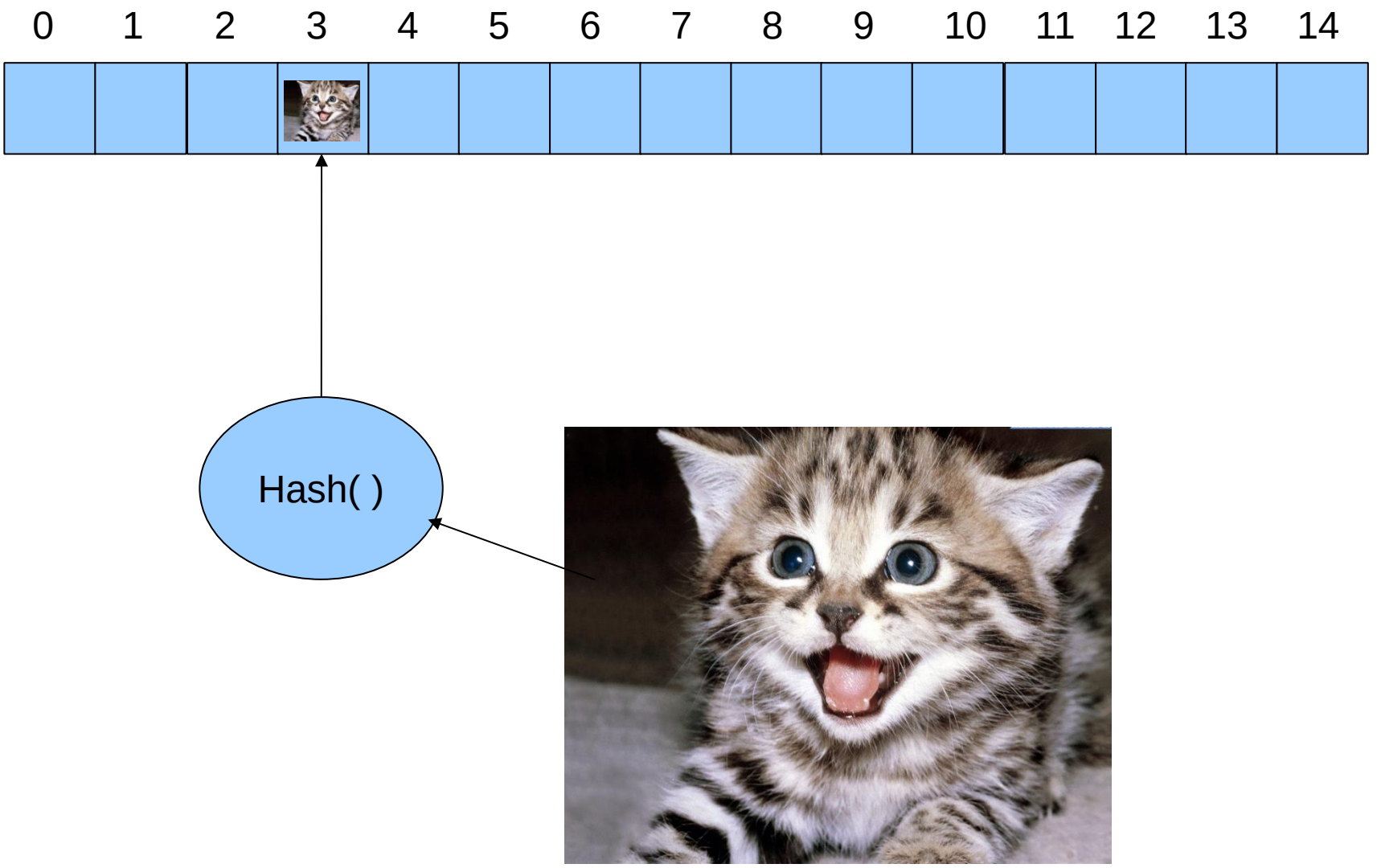


Array ad indirizzamento diretto



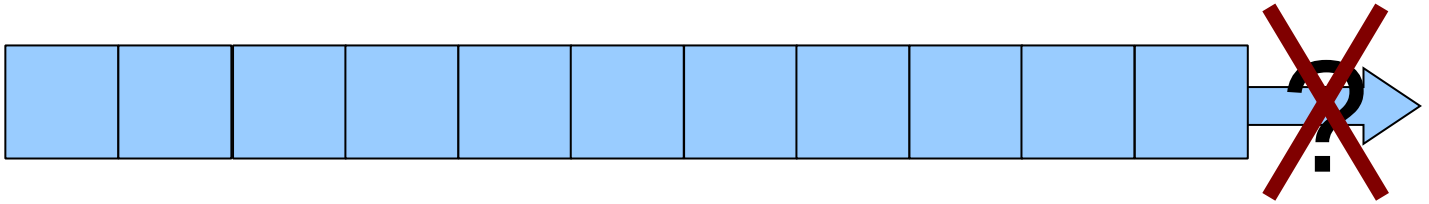
HASHING

Hashing



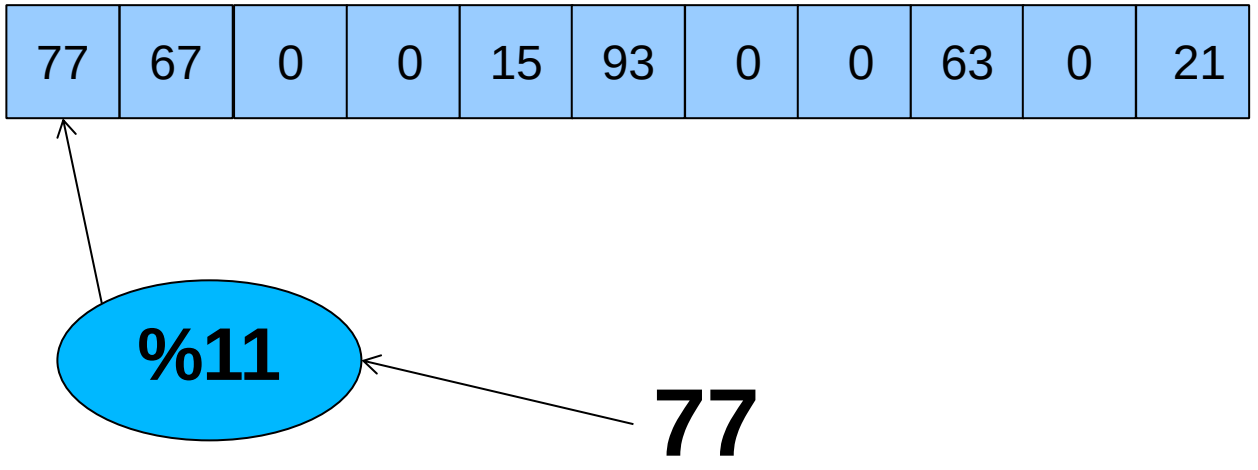
Strutture Dati

- Array
- Vector
- ...



Simple Hash Table

- Dati in input: interi positivi
- Chiave coincide con valore
- La funzione HASH e' la funzione modulo
- Convenzione: 0 per locazione vuota



Class

```
1   class HashTable
2   {
3       int * table_;
4       int size_;
5
6   public:
7       HashTable( int size );
8
9       bool insert( int key );
10
11      void print();
12
13      int hash( int key );
14  };
15
16
17
18
```

Costruttore

```
1  HashTable::HashTable( int size )
2  {
3      table_ =  new int[size];
4
5      size_ = size;
6
7
8
9  }
10
11
12
13
14
15
16
17
18
```

memset(address , value , size)

Example

```
1  /* memset example */
2  #include <stdio.h>
3  #include <string.h>
4
5  int main ()
6  {
7    char str[] = "almost every programmer should know memset!";
8    memset (str,'-',6);
9    puts (str);
10   return 0;
11 }
```

Output:

```
------ every programmer should know memset!
```

Hashing

```
1   int HashTable::hash( int key )
2   {
3
4       return key % size_;
5
6   }
7
8
9
10
11
12
13
14
15
16
17
18
```

Insert

```
1   bool HashTable::insert( int key )
2   {
3       // trova indice tramite hashing
4
5       // se posizione già occupata
6       {
7           // non posso inserire
8
9       }
10
11
12      // inserisco
13
14  }
15
16
17
18
```

Insert

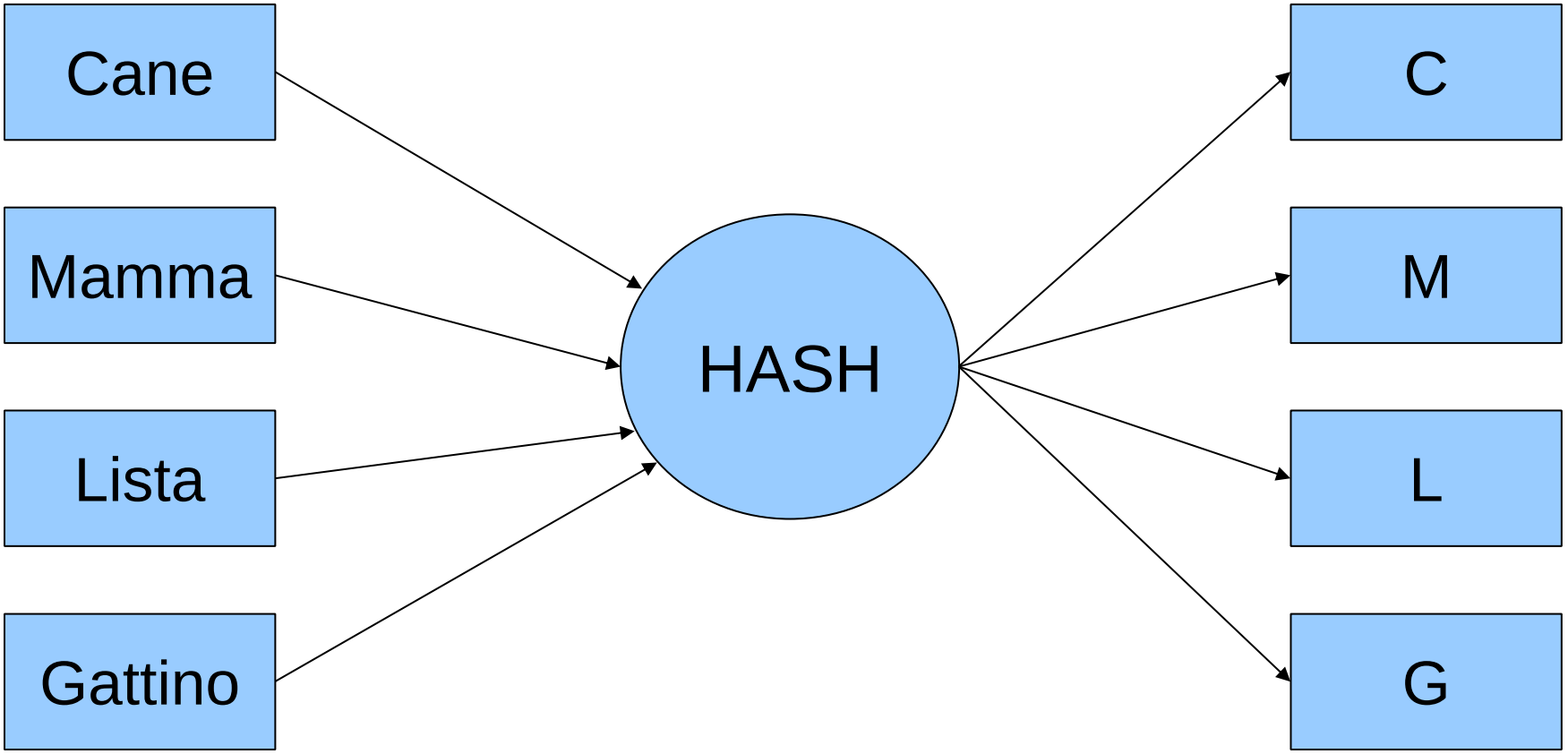
```
1   bool HashTable::insert( int key )
2   {
3       int index = hash(key);
4
5       if( table_[index] != 0 )
6       {
7           cout << "already occupied" << endl;
8           return false;
9       }
10      table_[index] = key;
11
12      cout << "key stored" << endl;
13      return true;
14  }
15
```

Esempio Hashing Stringhe: Prima lettera

```
int hash(string key)
{
    int index = key[0] % size_;
}
```

?

Esempio Hashing Stringhe: Prima lettera



Esempio Hashing Stringhe: Somma caratteri

```
for( int i = 0 ; i < key.length() ; ++i )
{
  index = ( index + key[i] )% size_;
}
```

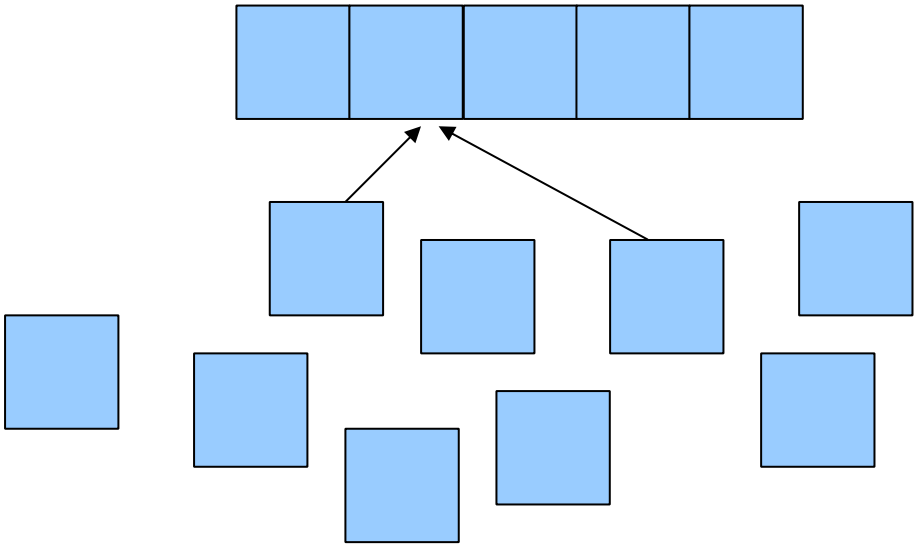
?

Good HASH

- Dipende fortemente dal tipo di applicazione
- Applicazioni con crittografia richiedono (tra le altre cose) di essere **difficilmente invertibili**.
- Per applicazioni di indexing e' fondamentale l'**uniformità**
- Lavorano sulla **rappresentazione binaria**
- E.g. MurmurHash,

  CityHash, FarmHash …

Already occupied? -> Collisione

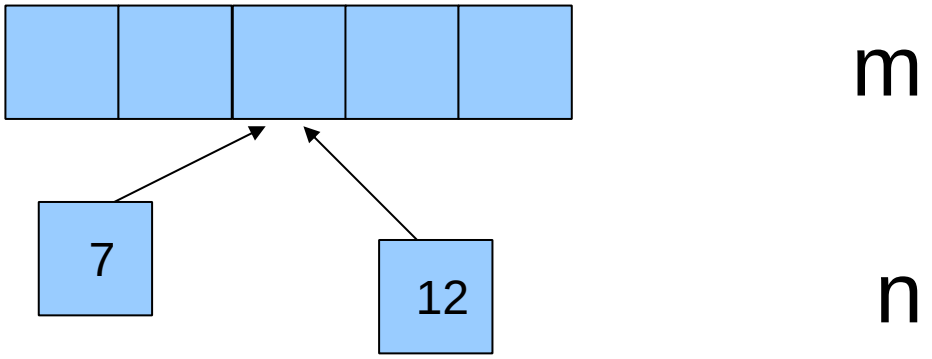


m

n

m<n

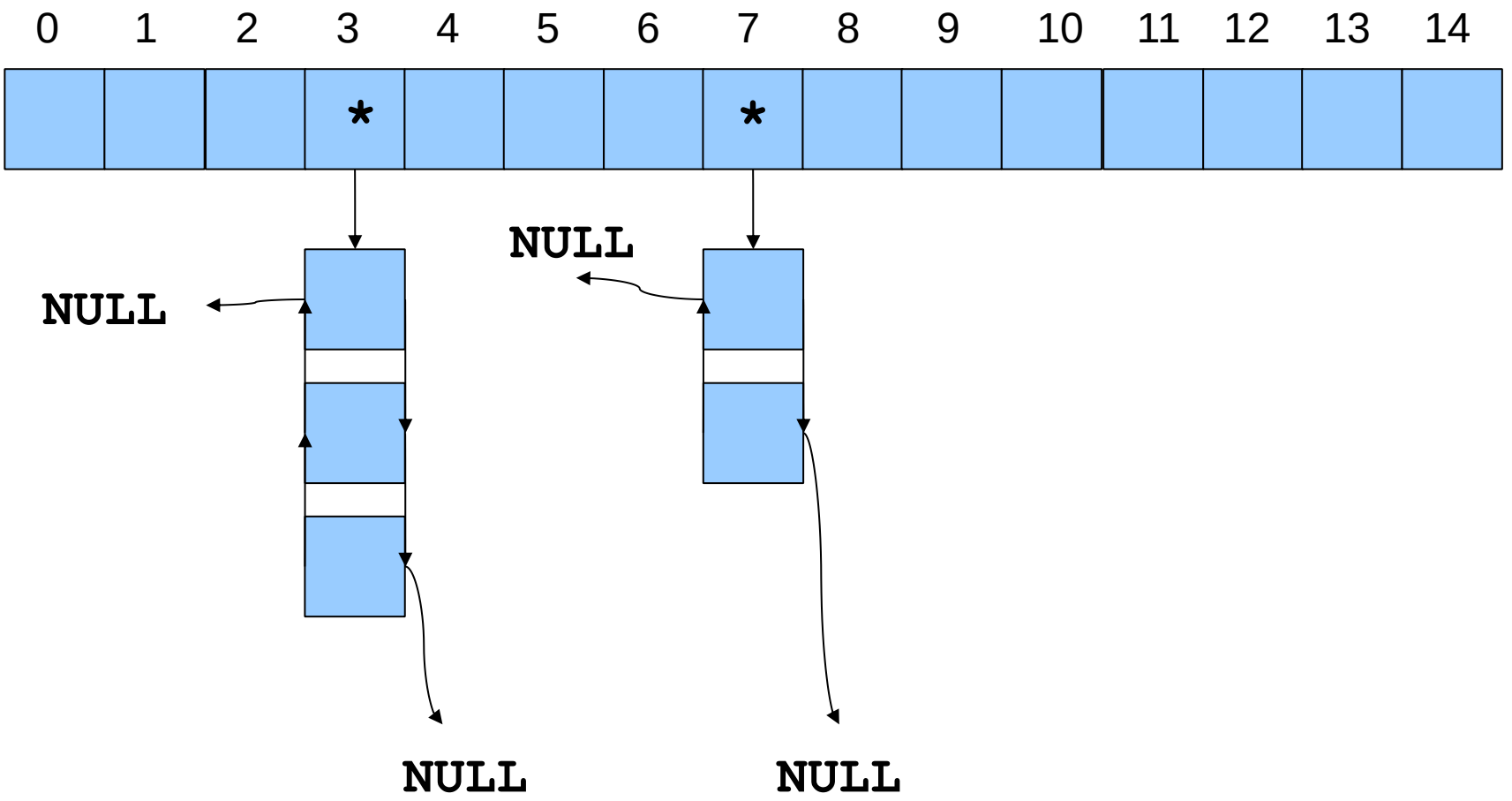
Gestione delle Collisioni



- Liste di trabocco

- Indirizzamento aperto

Array di puntatori

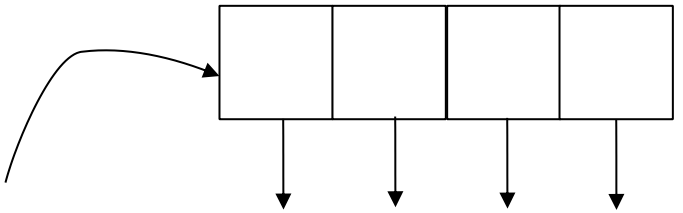


Elem

```
1  struct Elem
2  {
3     int key;
4     Elem * next;
5     Elem * prev;
6
7
8     Elem(): next(NULL) , prev(NULL) {}
9  };
10
11
12
13
14
15
16
17
18
```

Hash con trabocco

```
1   class HashTable
2   {
3       Elem * * table_;
4       int size_;
5
6
7   public:
8
9
10
11
12
13
14
15
16
17
18  };
```



Implementazione

- Insert / Print / Find

- Stiamo trattando liste con inserimento in testa

Indirizzamento Aperto

Si cerca nello slot prescelto, e poi negli slot “alternativi” a seguire fino a quando non si trova la chiave oppure NULL.

Diversamente dalle liste di trabocco, non ci sono liste né elementi memorizzati all'esterno della tavola.

stl::map

```
std::map < key_T , obj_t > table;
```

Tipo chiave **Tipo dati**

```
table['uno']="valore uno";

table.find('uno');
```

E infine…

Esperimenti

- Provare implementazioni variando:
  - Test dimensione table
  - Test tipi di hash
- Effettuare Confronto map vs hash
- Effettuare Confronto tempi di esecuzione Sorting e tabelle hash
- Interrogazioni:
  - Conoscere i primi K
  - Conoscere posizione di uno specifico elemento

Esercizio

Si consideri un sistema di memorizzazione dei dati relativi a dei veicoli. Ogni veicolo è rappresentato da un valore *targa* univoco, intero e positivo, e da un valore intero *categoria* compreso tra 0 e $C-1$. Il sistema legge i dati relativi a N veicoli e li inserisce dentro una *tabella hash*, utilizzando la *targa* come etichetta. La tabella hash è realizzata con il metodo di concatenazione.

Per ogni indice i della tabella hash si definisce $M(i)$ come la categoria con più veicoli in i, e $V(i)$ come il numero di veicoli in i appartenenti a $M(i)$. A parità di numero di veicoli, si considerino le *categorie* in ordine crescente. Scrivere un programma che:

- legga da tastiera una sequenza di N coppie di interi {*targa*,*categoria*} e le inserisca nella tabella hash all'indirizzo dato dalla seguente funzione hash:

$$h(x) = \{[(\ a \times x\ ) +\ \ b\ ]\ \%\ p\ \}\ \%\ (C)$$

  dove p=999149 , a=1000 e b=2000;

- Stampi a video i primi K indirizzi i della tabella hash in ordine di $V(i)$ decrescente. A parità di $V(i)$, si considerino gli indirizzi in ordine crescente.

L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N, K e C separati da uno spazio. Seguono N righe contenenti una coppia di interi ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
11 6 7
170 1
96 2
577 0
692 0
873 6
446 6
732 1
394 1
845 1
715 1
456 3
```

Output

```
0
3
1
2
4
6
```

Possibile soluzione

```
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector
#include <fstream>
#include <math.h>       /* floor */
#include <stdlib.h>
#include <cmath>        /* pow  */

using namespace std;


struct Veicolo
{
    int targa;
    int categoria;

    Veicolo(): targa(-1) , categoria(-1) {};
    Veicolo( int t , int c ): targa(t) , categoria(c) {}
};

// conterrà le info relative a ciascuna entrata della tabella hash
struct Info
{
    int indirizzo;
    int veicoli;
};
```

```
class Concatenazione
{
    vector<int> categorie_;
    vector<Veicolo> veicoli_;

public:
    Concatenazione( int C )
    {
        categorie_.resize(C, 0);
    }

    // inserisci e aggiorna l'utente max
    void inserisci( Veicolo v )
    {
        veicoli_.push_back(v);

        ++categorie_[v.categoria];
    }

    int getMaxV()
    {
        int vMax = -1;
        int c;

        for( c = 0 ; c < categorie_.size(); ++c )
        {
//              cout << "\t" << c << "," << categorie_[c] << endl;
            if( categorie_[c] > vMax )
            {
                vMax = categorie_[c];
            }
        }
        return vMax;
    }
};
```

```
bool compare( Info a , Info b )
{
    if( a.veicoli > b.veicoli )
        return true;
    else if( a.veicoli == b.veicoli)
        return a.indirizzo < b.indirizzo;
    else
        return false;
}

int main()
{
    unsigned int N , K , C;
    int x , c ;

    cin >> N >> K >> C;
    Concatenazione cBase(C);
    vector<Concatenazione> hashTable(C, cBase);

    int pos;

    Veicolo tempVeicolo;
    // riempio le liste di concatenazione
    for(unsigned int i=0 ; i<N ; ++i )
    {
        // leggo e creo un nuovo utente
        cin >> x >> c;
        tempVeicolo.targa = x;
        tempVeicolo.categoria = c;

        pos = hashFun( x , C );

//          cout << x << "," << c << " -> " << pos << endl;

        hashTable[pos].inserisci(tempVeicolo);
    }
```

```
    // calcolo insieme I
    vector <Info> info;
    Info tempInfo;
    for( unsigned int i=0 ; i<C ; ++i )
    {
        tempInfo.indirizzo = i;

//          cout << endl << "============== " << i << " ==============" << endl;
        tempInfo.veicoli = hashTable[i].getMaxV();
        info.push_back( tempInfo );
    }
    sort( info.begin() , info.end() , compare );

    for( unsigned int i=0 ; i < K && i<info.size() ; ++i )
    {
        cout << info[i].indirizzo << endl;
    }
}
```