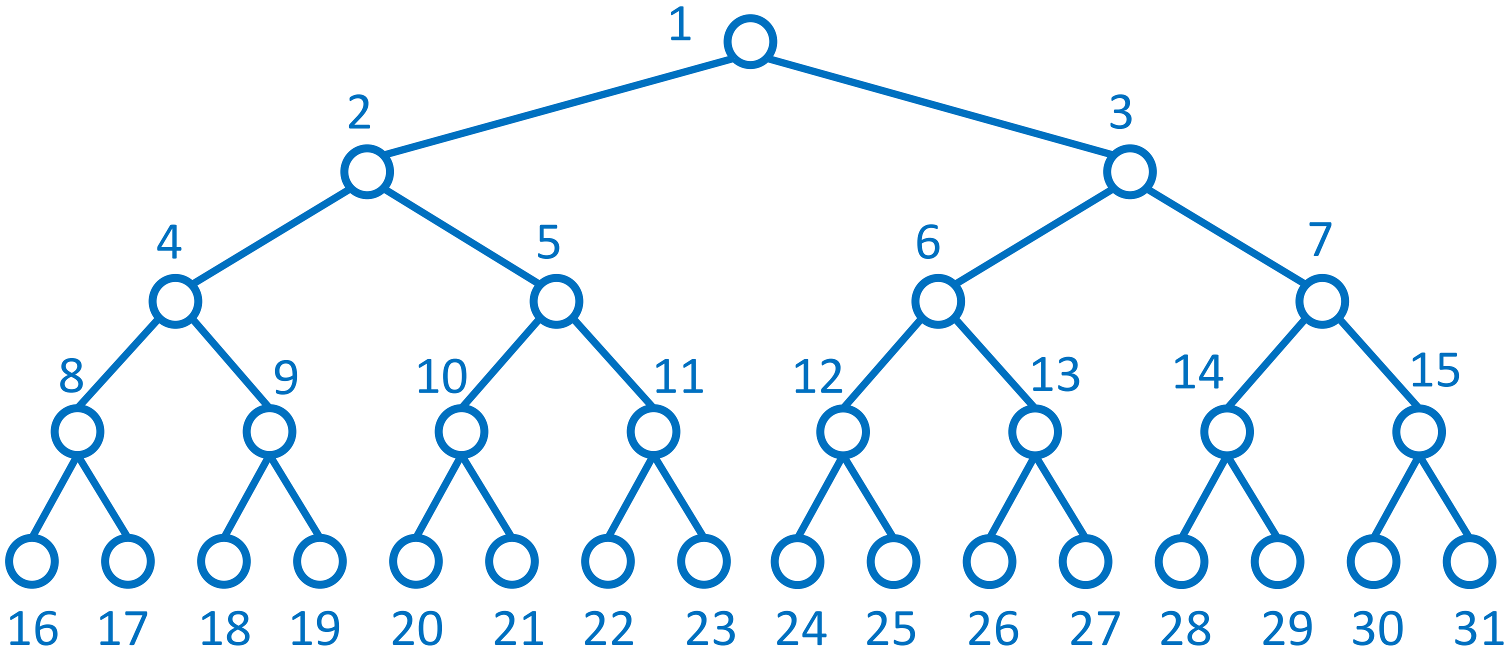
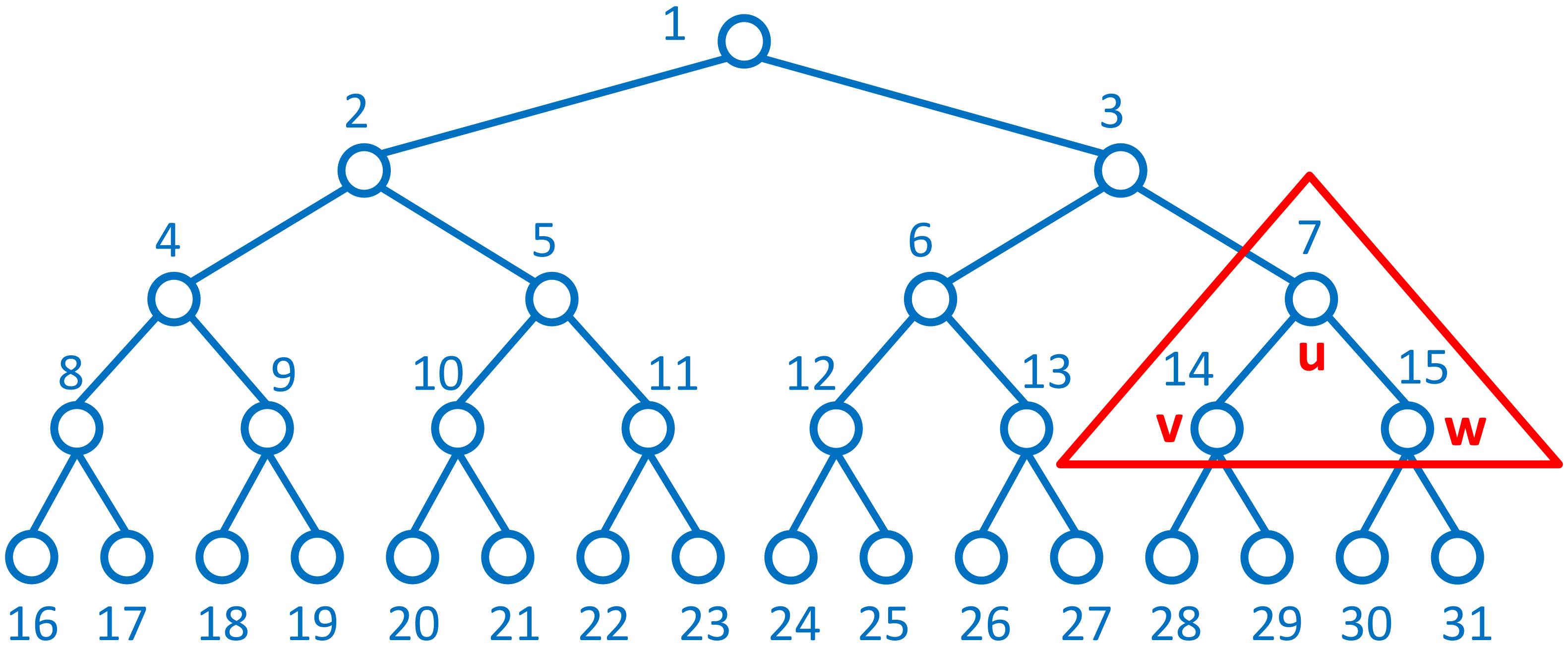
BINARY TREE – VECTOR IMPLEMENTATION

- Given a binary tree, T, let **f(v)** be the integer defined as follows:
  - If v is the **root** of T, then **f(v) = 1**
  - If v is the **left child** of node u, then **f(v) = 2 f(u)**
  - If v is the **right child** of node u, then **f(v) = 2 f(u) + 1**
- The function **f** is known as a **level numbering** of the nodes in a binary tree T



BINARY TREE – VECTOR IMPLEMENTATION

- Given a binary tree, T, let **f(v)** be the integer defined as follows:
  - If v is the **root** of T, then **f(v) = 1**
  - If v is the **left child** of node u, then **f(v) = 2 f(u)**
  - If v is the **right child** of node u, then **f(v) = 2 f(u) + 1**
- The function **f** is known as a **level numbering** of the nodes in a binary tree T



Example:

Consider nodes u, v, w

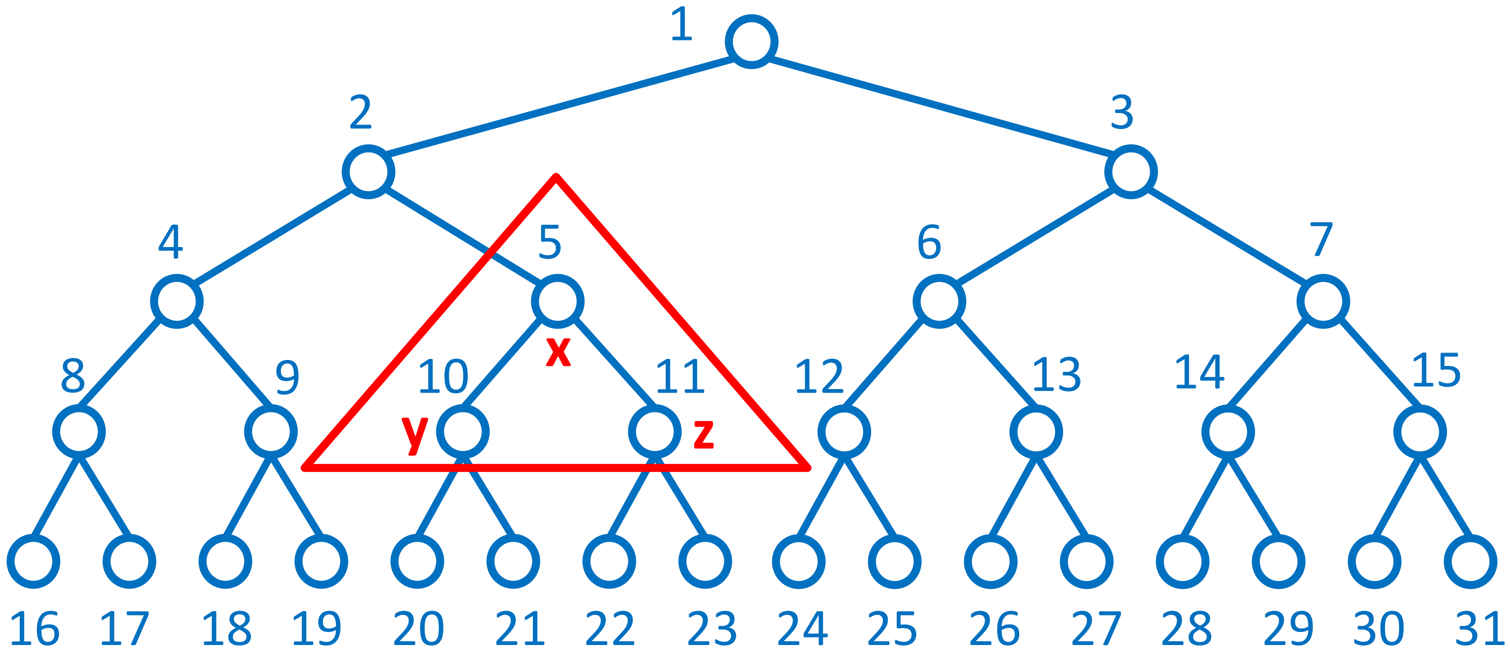
f(u) = 7

f(v) = 14 = 2 f(u)

f(w) = 15 = 2 f(u) + 1

BINARY TREE – VECTOR IMPLEMENTATION

- Given a binary tree, T, let **f(v)** be the integer defined as follows:
  - If v is the **root** of T, then **f(v) = 1**
  - If v is the **left child** of node u, then **f(v) = 2 f(u)**
  - If v is the **right child** of node u, then **f(v) = 2 f(u) + 1**
- The function **f** is known as a **level numbering** of the nodes in a binary tree T



Example:

Consider nodes x, y, z

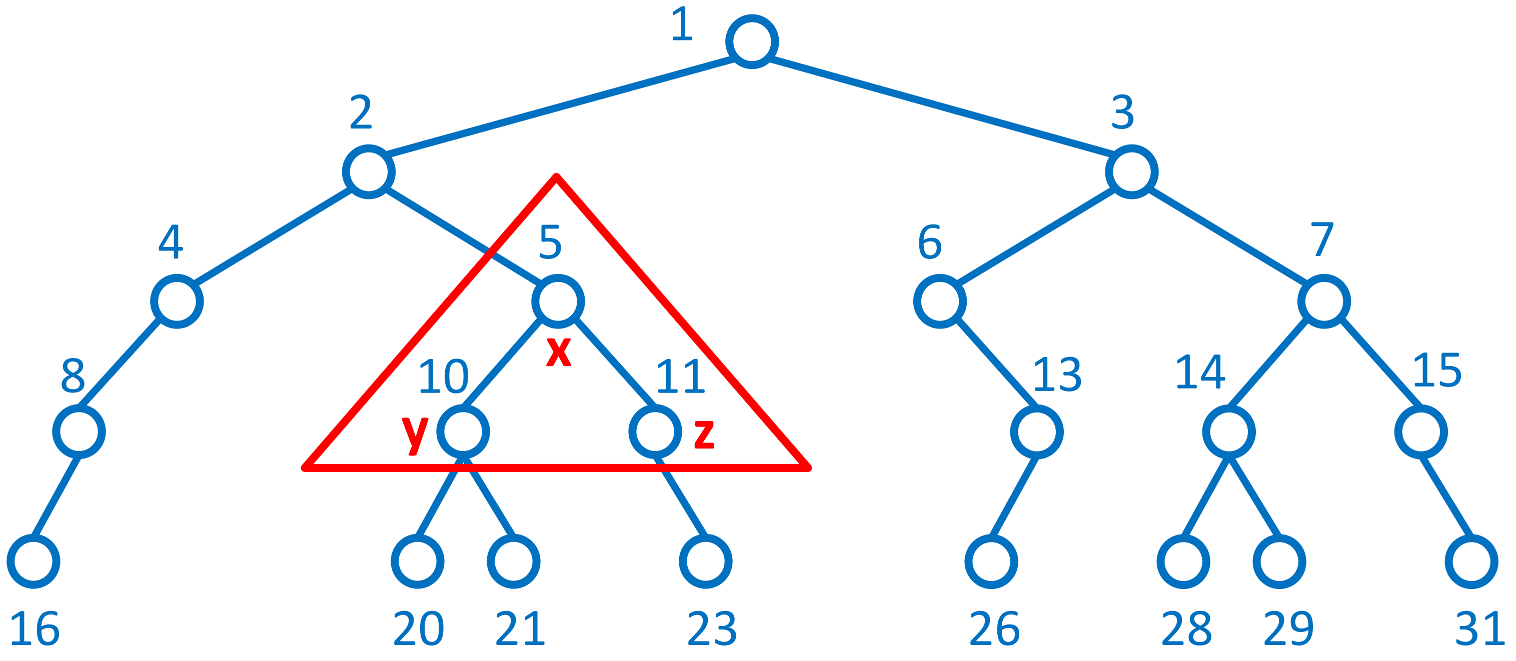
f(x) = 5

f(y) = 10 = 2 f(x)

f(z) = 11 = 2 f(x) + 1

BINARY TREE – VECTOR IMPLEMENTATION

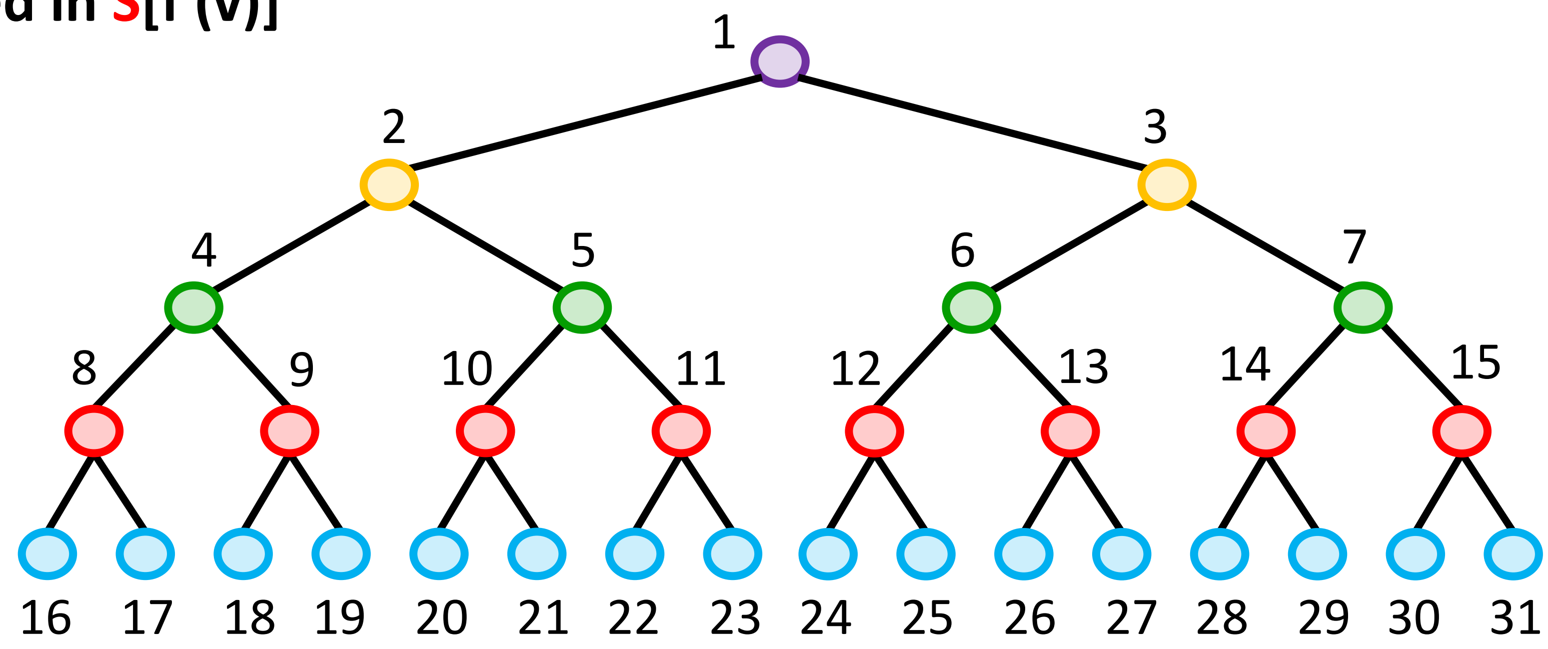
- Given a binary tree, T, let **f(v)** be the integer defined as follows:
  - If v is the **root** of T, then **f(v) = 1**
  - If v is the **left child** of node u, then **f(v) = 2 f(u)**
  - If v is the **right child** of node u, then **f(v) = 2 f(u) + 1**
- The function **f** is known as a **level numbering** of the nodes in a binary tree T

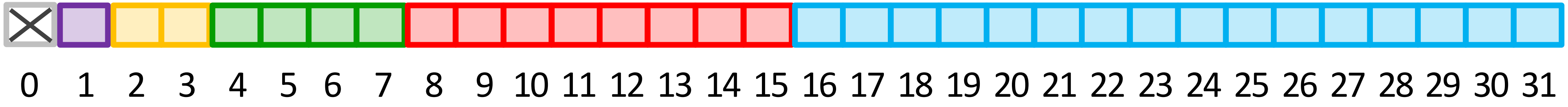


Note that the level numbering remains the same even if some nodes are missing from different levels!

BINARY TREE – VECTOR IMPLEMENTATION

- Using the level numbering, we can represent T as a **vector**, S, such that **node v of T is stored in S[f (v)]**

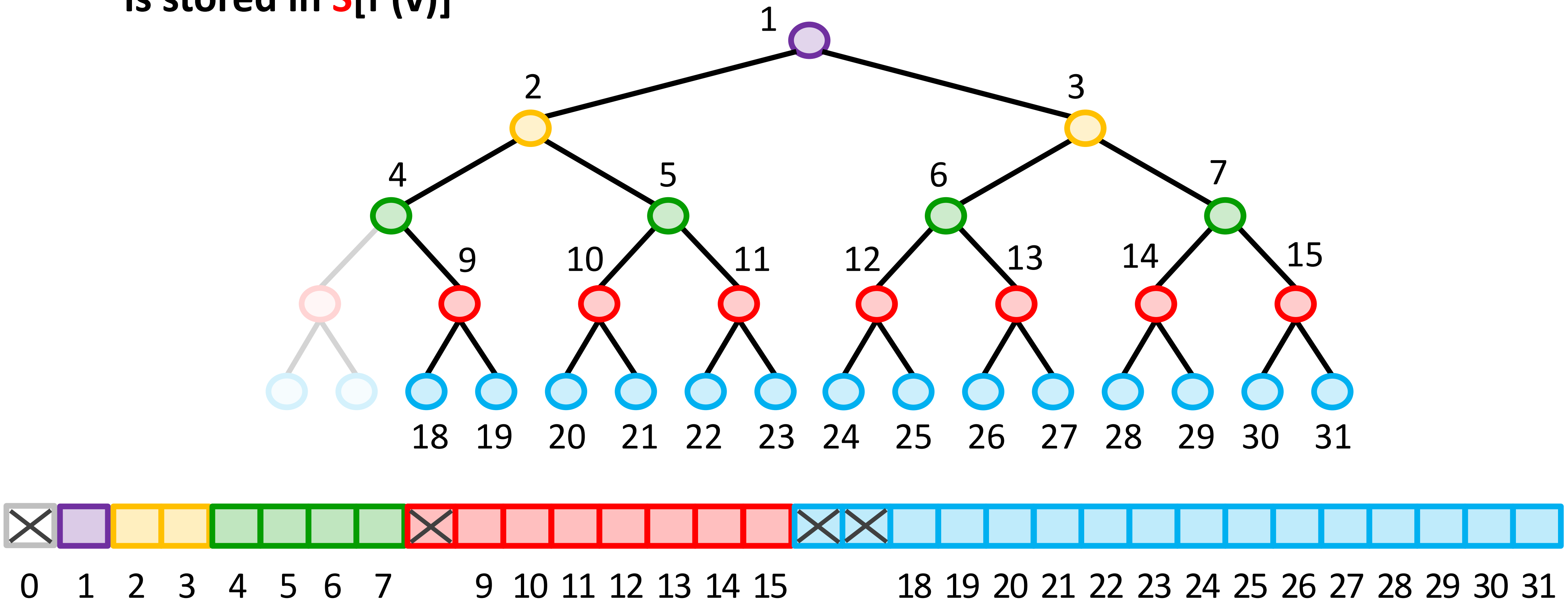




What could be a disadvantage of the vector based implementation?

BINARY TREE – VECTOR IMPLEMENTATION

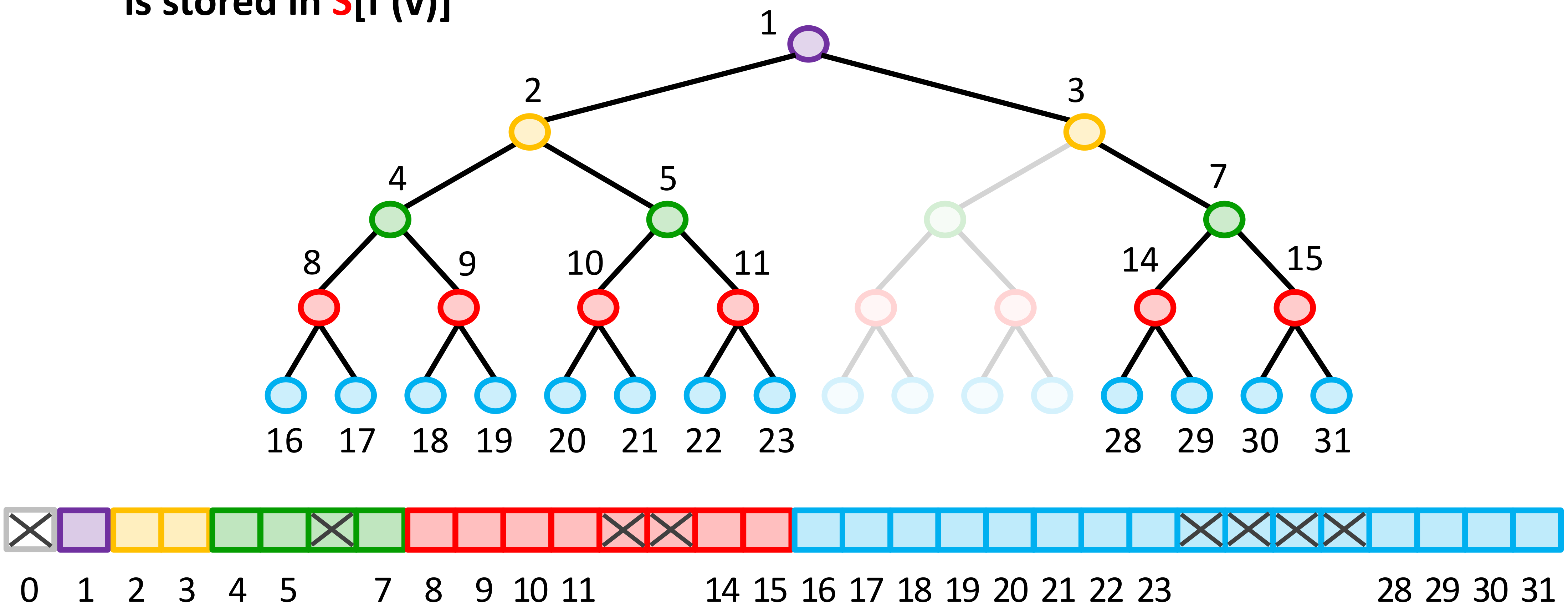
- Using the level numbering, we can represent T as a **vector**, S, such that **node v of T is stored in S[f (v)]**



- **Disadvantage: We waste memory space if some nodes are missing!**

BINARY TREE – VECTOR IMPLEMENTATION

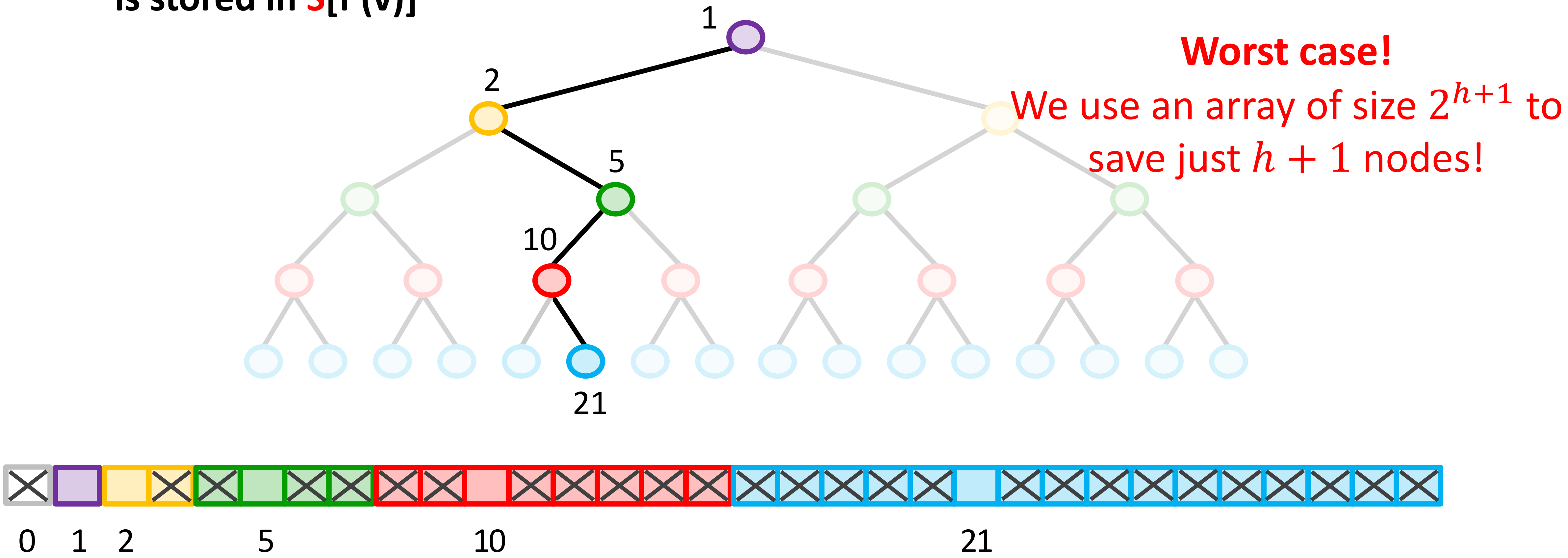
- Using the level numbering, we can represent T as a **vector**, S, such that **node v of T is stored in S[f (v)]**



- **Disadvantage: We waste memory space if some nodes are missing!**

BINARY TREE – VECTOR IMPLEMENTATION

- Using the level numbering, we can represent T as a **vector**, S, such that **node v of T is stored in S[f (v)]**

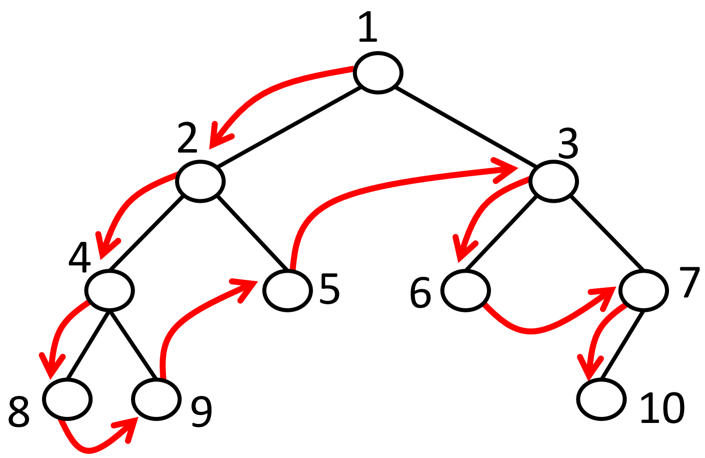


- **Disadvantage: We waste memory space if some nodes are missing!**

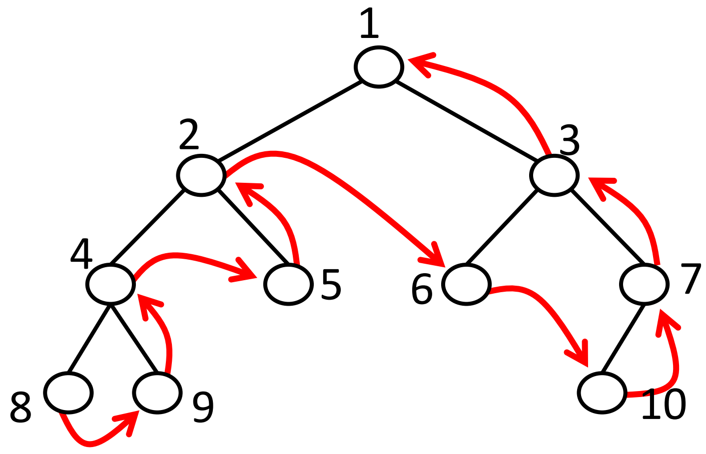
BINARY TREE TRAVERSAL

- This is the same as for general trees, except that we now make a recursive call for the left child and the right child instead of making it for all children

Algorithm binaryPreorder(T, p):
    perform the "visit" action for node p
    if p is an internal node then{
        binaryPreorder(T, p.left())
        binaryPreorder(T, p.right())
    }



Algorithm binaryPostorder(T, p):
    if p is an internal node then{
        binaryPostorder(T, p.left())
        binaryPostorder(T, p.right())
    }
    perform the "visit" action for node p



BINARY TREE TRAVERSAL

- This is the same as for general trees, except that we now make a recursive call for the left child and the right child instead of making it for all children

```
Algorithm binaryPreorder(T, p):
    perform the "visit" action for node p
    if p is an internal node then{
        binaryPreorder(T, p.left())
        binaryPreorder(T, p.right())
    }
```

```
Algorithm binaryPostorder(T, p):
    if p is an internal node then{
        binaryPostorder(T, p.left())
        binaryPostorder(T, p.right())
    }
    perform the "visit" action for node p
```

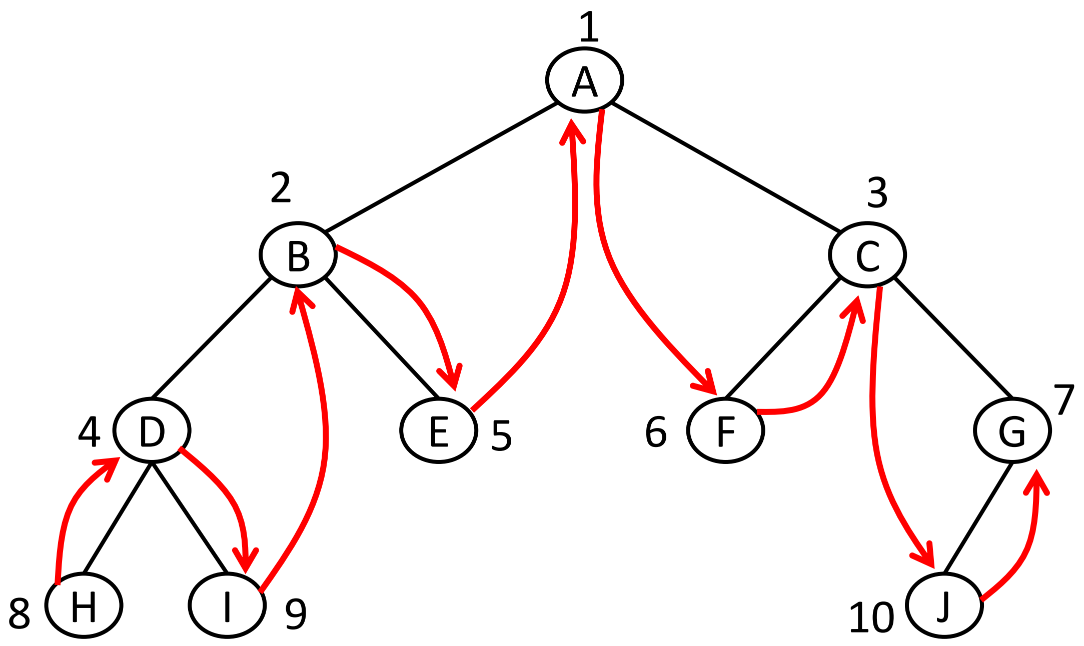
For binary trees, we can also use In-order traversal

```
Algorithm Inorder(T, p):
    if p is an internal node then
        Inorder(T, p.left())
    perform the "visit" action for node p
    if p is an internal node then
        Inorder(T, p.right())
```

BINARY TREE TRAVERSAL

- This is the same as for general trees, except that we now make a recursive call for the left child and the right child instead of making it for all children

How do you think in-order works?



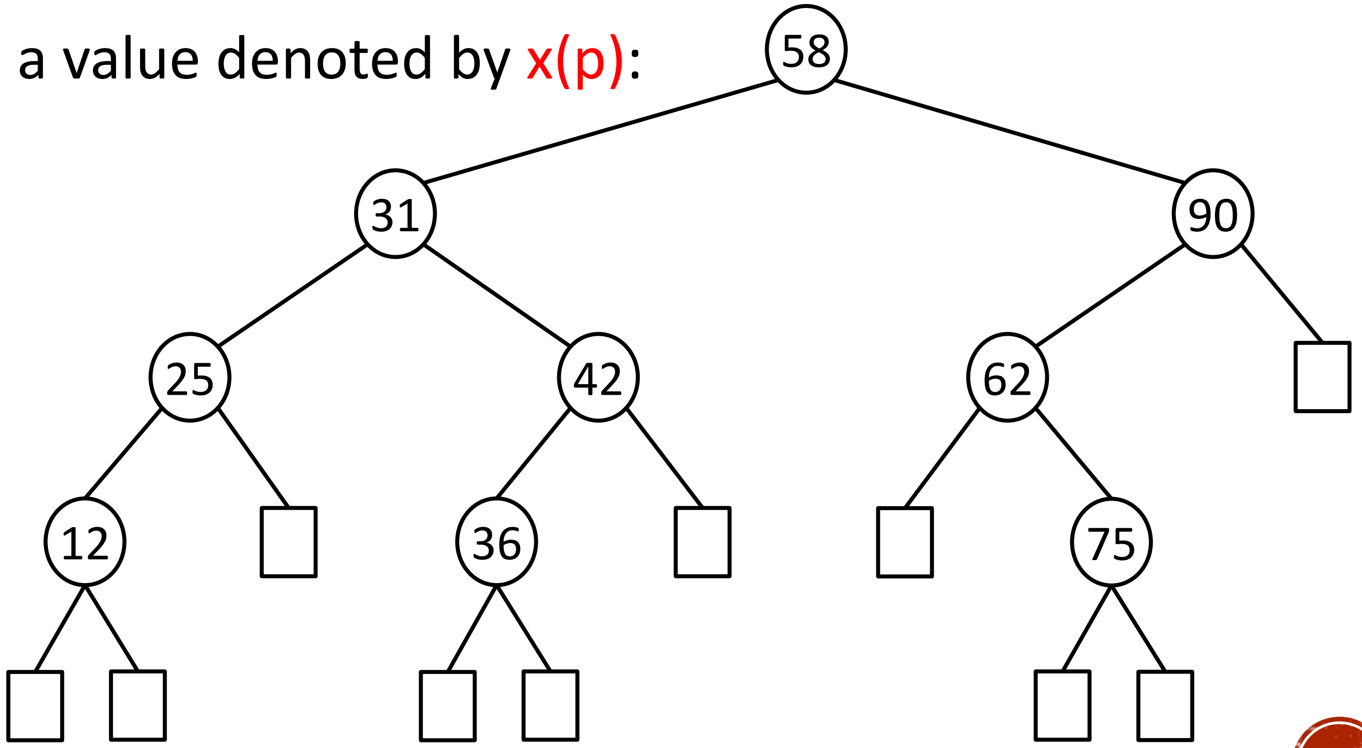
For binary trees, we can also use In-order traversal

Algorithm Inorder(T, p):
    if p is an internal node then
        Inorder(T, p.left())
    perform the “visit” action for node p
    if p is an internal node then
        Inorder(T, p.right())

BINARY SEARCH TREE

A *binary search tree (BST)* is a **proper binary tree** such that:

- **External nodes** (represented as squares) do not store elements
- Each **internal node** p stores an object of some type
- For each **internal node** p, there is a value denoted by x(p):
  - Any object stored in the **left subtree** of p has such a value that is $\leq$ x(p)
  - Any object stored in the **right subtree** of p has such a value that is $\geq$ x(p)

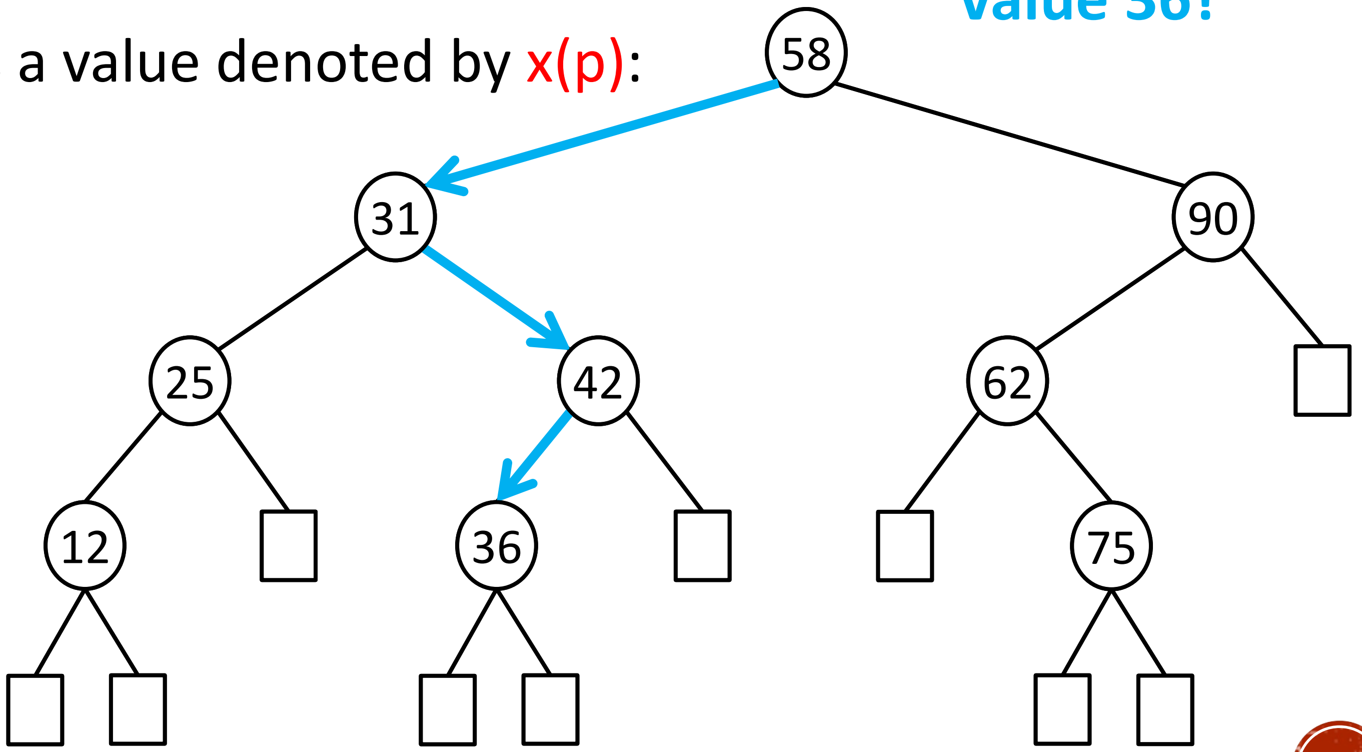


BINARY SEARCH TREE

A *binary search tree (BST)* is a **proper binary tree** such that:

- **External nodes** (represented as squares) do not store elements
- Each **internal node** p stores an object of some type
- For each **internal node** p, there is a value denoted by x(p):
  - Any object stored in the **left subtree** of p has such a value that is $\leq$ x(p)
  - Any object stored in the **right subtree** of p has such a value that is $\geq$ x(p)

How do we search for a node with the value 36?

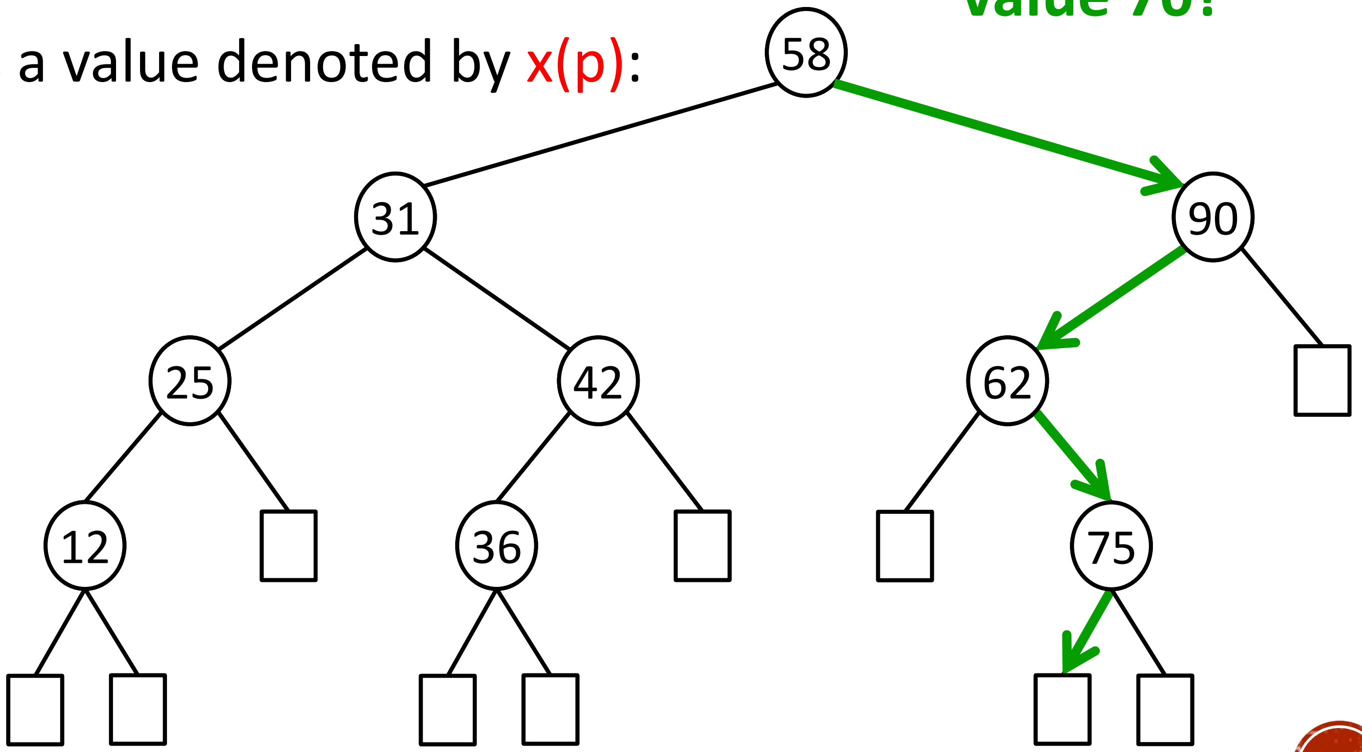


BINARY SEARCH TREE

A *binary search tree (BST)* is a **proper** **binary tree** such that:

- **External nodes** (represented as <u>squares</u>) do not store elements
- Each **internal node** p stores an object of some type.
- For each **internal node** p, there is a value denoted by x(p):
  - ➢ Any object stored in the **left subtree** of p has such a value that is $\leq$ x(p)
  - ➢ Any object stored in the **right subtree** of p has such a value that is $\geq$ x(p)

How do we search for a node with the value 70?



CHAPTER 8:
HEAPS & PRIORITY QUEUES

PRIORITY QUEUE

PRIORITY QUEUE

- A priority queue is a data structure, where each element is associated with a **key** that represents its priority (i.e. its importance or weight)

- This is **fundamentally different from the position-based data structures** such as stacks, queues, deques, lists, and trees, all of which store elements at specific positions
  - A priority queue has no notion of position

- In a priority queue, **more than one element can have the same key**

- Note that **a key does not have to be a number,** it can be of any type, and can even be an object of a class!

TOTAL ORDER

- A priority queue needs a comparison rule that can compare any two keys to determine which one is “smaller”.

- Such a comparison rule **must never contradict itself**. To achieve this, it must satisfy these properties:

  - **Reflexive property** : $k \leq k$
  - **Antisymmetric property**: if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
  - **Transitive property**: if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$

- Any rule, $\leq$, that satisfies these properties never leads to a comparison contradiction. In this case, we say that it defines a *total order* relation.

COMPARATOR

- We need to implement the comparison rule as a **function**. This function will be referred to as the "**comparator**".

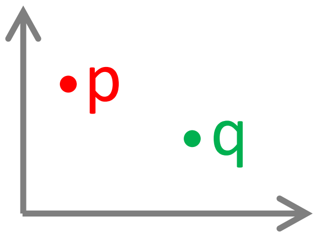
Example:

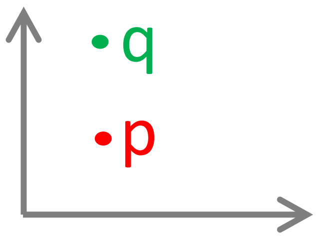
- Suppose we have a class called "Point2D" that represents a 2-dimentional point, where the methods `getX()` and `getY()` return the x and y coordinates.

- **One possible comparator could be:**

```
bool isLess(const Point2D& p, const Point2D& q) {
    if (p.getX() == q.getX())   return p.getY() < q.getY();
                       else     return p.getX() < q.getX();
}
```

- Here are two examples where p is "less than" q.
(the point that is more to the left is "smaller"; if both are equally to the left, the bottom one is "smaller")





COMPARATOR – OVERLOADING "<"

- It is practical to overload the operator "<" so that instead of writing, e.g.,

  ```
  if( isLess( p, q ) ) then . . .
  ```

  we can simply write:

  ```
  if( p < q ) then . . .
  ```

- To do so, instead of defining the comparator as follows:

  ```
  bool isLess(const Point2D& p, const Point2D& q) {
      if (p.getX() == q.getX())   return p.getY() < q.getY();
                          else    return p.getX() < q.getX();
  }
  ```

  we can define it as:

  ```
  bool operator<(const Point2D& p, const Point2D& q) {
      if (p.getX() == q.getX())   return p.getY() < q.getY();
                          else    return p.getX() < q.getX();
  }
  ```

MULTIPLE COMPARATORS

- In some applications, we may need to define multiple, different comparators!

Example:

- When comparing 2-dimentional points, we might require two comparators:

```
// The point more to the left is considered “smaller”
bool leftRight(const Point2D& p, const Point2D& q) {
    return p.getX() < q.getX();
}

// The point more to the bottom is considered “smaller”
bool bottomTop (const Point2D& p, const Point2D& q) {
    return p.getY() < q.getY();
}
```

MULTIPLE COMPARATORS

How can we disguise our comparator functions as types?

- We'll make each comparator function appear as a class! To do this, simply take each function, and make it a method in a class dedicated for that function!

```
bool leftRight(Point2D p, Point2D q) {
    return p.getX() < q.getX();
}
```

⇩

```
Class LeftRight {
public:
    bool operator()(Point2D p, Point2D q) const {
        return p.getX() < q.getX();
    }
}
```

```
bool bottomTop (Point2D p, Point2D q) {
    return p.getY() < q.getY();
}
```

⇩

```
Class BottomTop {
public:
    bool operator()(Point2D p, Point2D q) const {
        return p.getY() < q.getY();
    }
}
```

MULTIPLE COMPARATORS

- Now that our comparator functions have become classes (i.e., types), we can pass them to templates! Instead of repeating code that calls different comparators:

```
void function_1 ( int x, int y ){
    /* code goes here that
       uses comparator_1 */
}
```

```
void function_2 ( int x, int y ){
    /* The same code goes here, but
       uses comparator_2 */
}
```

```
void function_3 ( int x, int y ){
    /* The same code goes here, but
       uses comparator_3 */
}
```

We can define a single template that take a type representing the comparators:

```
template <typename T>  void myFunction( int x, int y, T comparator) {
    // code goes here that uses comparator
}
```

MULTIPLE COMPARATORS

- Example: Instead of defining 2 versions of a function that print the "smallest" point

```
void printLeft( Point2D p, Point2D q) {
    if( leftRight( p, q) ) cout << p;
                    else  cout << q;
}
```

```
void printBottom( Point2D p, Point2D q) {
    if( BottmTop( p, q) ) cout << p;
                    else  cout << q;
}
```

We can define a single template that takes a type representing the comparators:

```
template <typename T>  void printSmaller( Point2D p, Point2D q, T isLess) {
    if( isLess( p, q) ) cout << p;
                  else  cout << q;
}
```

- Here is how we can use such a template:

```
Point2D  p(2, 6),  q(3, 1);      // define two points, p and q
LeftRight  c1;                   // c1 is a function object representing a "LeftRight" comparator
BottomTop  c2;                   // c2 is a function object representing a "BottomTop" comparator
printSmaller( p, q, c1 );        // outputs: (2, 6)
printSmaller( p, q, c2 );        // outputs: (3, 1)
```

COMPOSITION METHOD

- As we said, a comparator **takes two elements** and determines which is "smaller"
- This implies that there is something (i.e. attributes) in those elements that allows the comparator to determine their **keys**, to decide which one is "smaller"
- But there are applications where **the key is independent from the element**
  - Example: a hotel may have a record for each client, and the room key of a client cannot be determined solely based on that client's attributes!
- In such applications, **we can't use comparators!** Instead, we'll have an object with two members (1) key; (2) element, e.g., (1) the room key; (2) the client's record.
- This idea to separate the key from the element is called the ***composition method***, which will be discussed in **Chapter 9**.

PRIORITY QUEUE ADT

A priority queue ADT P supports these functions:

- `size()`: Return the **number of elements** in P.
- `empty()`: Return **true if** P **is empty** and false otherwise.
- `insert(e)`: **Insert** e into P.
- `min()`: Return a reference to **an element of** P **with the smallest associated key value** (but do not remove it); ERROR if the priority queue is empty.
- `removeMin()`: **Remove** from P **the element referenced by** `min()`; ERROR if the priority queue is empty.

Note: more than one element can have the same key, which is why we define `removeMin()` to *remove not just any minimum element, but the same element returned by* `min()`.

Note: We don't specify how e is inserted into P. Depending on the implementation, it may be inserted **always at the end of the queue**, or inserted **based on its key** such that, after the insertion, the elements in P are sorted according to their keys

PRIORITY QUEUE – EXAMPLE

- In this is a priority queue:
  - It contains **integer elements**
  - The key of any element, *e*, is *e* itself!
  - Insertion is implemented such that, after an element is inserted, the priority queue is sorted!

- But don't forget that:
  - A priority queue **supports elements of any type**
  - The key associated with each element doesn't have to be the element itself!
  - Insertion may be implemented such that elements are always is inserted at the end!

Operation	***Output***	***Priority Queue***
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	"error"	{}