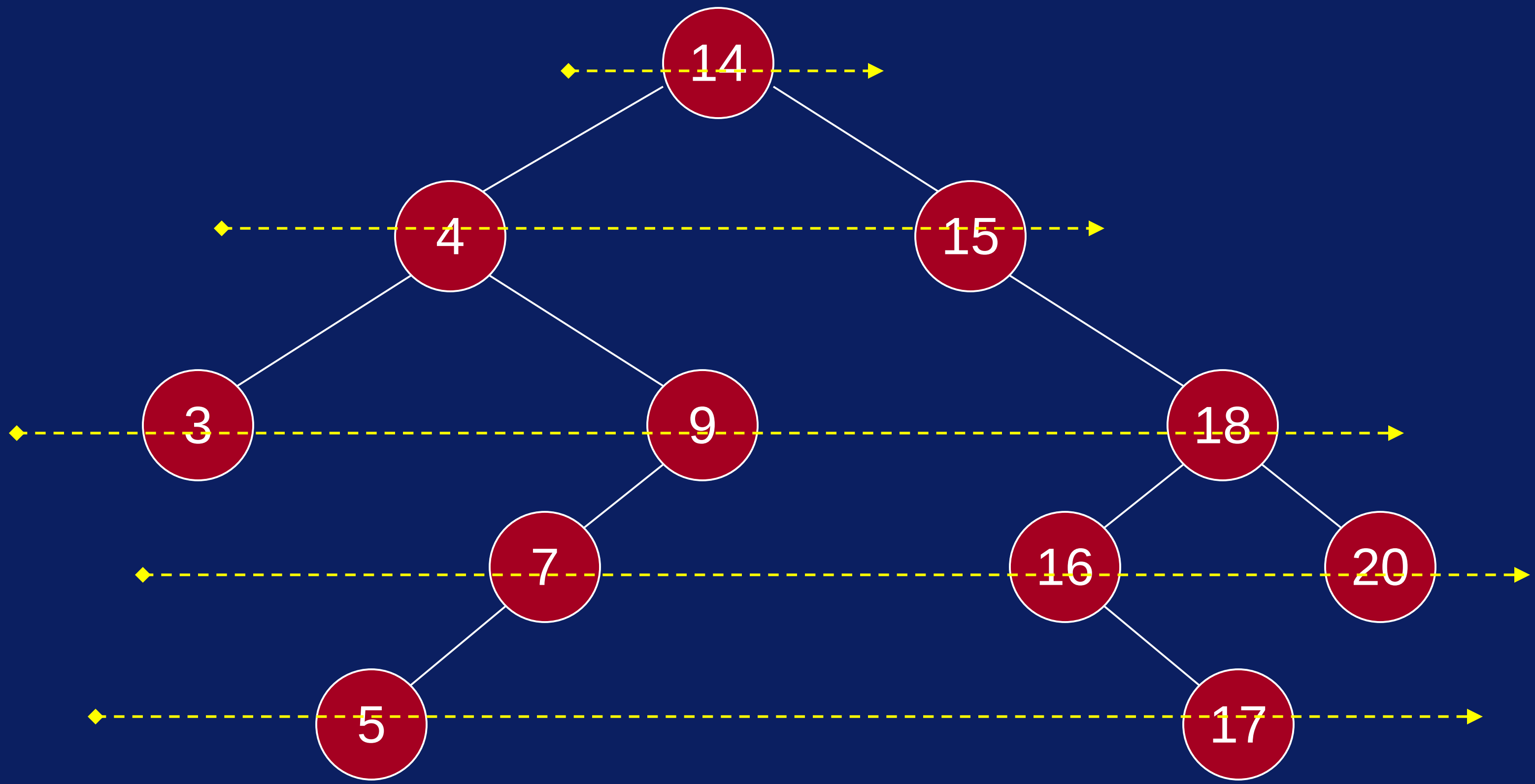
Lecture No.15

Data Structures

Level-order Traversal

- There is yet another way of traversing a binary tree that is not related to recursive traversal procedures discussed previously.
- In level-order traversal, we visit the nodes at each level before proceeding to the next level.
- At each level, we visit the nodes in a left-to-right order.

Level-order Traversal



Level-order: 14 4 15 3 9 18 7 16 20 5 17

Level-order Traversal

- How do we do level-order traversal?
- Surprisingly, if we use a queue instead of a stack, we can visit the nodes in level-order.
- Here is the code for level-order traversal:

Level-order Traversal

```
void levelorder(TreeNode<int>* treeNode)
{
    Queue<TreeNode<int>* > q;
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
        treeNode = q.dequeue();
        cout << *(treeNode->getInfo()) << " ";
        if(treeNode->getLeft() != NULL )
             q.enqueue( treeNode->getLeft());
        if(treeNode->getRight() != NULL )
             q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```

Level-order Traversal

```
void levelorder(TreeNode<int>* treeNode)
{
    Queue<TreeNode<int>* > q;
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
        treeNode = q.dequeue();
        cout << *(treeNode->getInfo()) << " ";
        if(treeNode->getLeft() != NULL )
             q.enqueue( treeNode->getLeft());
        if(treeNode->getRight() != NULL )
             q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```

Level-order Traversal

```
void levelorder(TreeNode<int>* treeNode)
{
    Queue<TreeNode<int>* > q;
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
        treeNode = q.dequeue();
        cout << *(treeNode->getInfo()) << " ";
        if(treeNode->getLeft() != NULL )
             q.enqueue( treeNode->getLeft());
        if(treeNode->getRight() != NULL )
             q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```

Level-order Traversal

```
void levelorder(TreeNode<int>* treeNode)
{
    Queue<TreeNode<int>* > q;
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
        treeNode = q.dequeue();
        cout << *(treeNode->getInfo()) << " ";
        if(treeNode->getLeft() != NULL )
             q.enqueue( treeNode->getLeft());
        if(treeNode->getRight() != NULL )
             q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```

Level-order Traversal

```
void levelorder(TreeNode<int>* treeNode)
{
    Queue<TreeNode<int>* > q;
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
        treeNode = q.dequeue();
        cout << *(treeNode->getInfo()) << " ";
        if(treeNode->getLeft() != NULL )
             q.enqueue( treeNode->getLeft());
        if(treeNode->getRight() != NULL )
             q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```

Level-order Traversal

```
void levelorder(TreeNode<int>* treeNode)
{
    Queue<TreeNode<int>* > q;
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
        treeNode = q.dequeue();
        cout << *(treeNode->getInfo()) << " ";
        if(treeNode->getLeft() != NULL )
             q.enqueue( treeNode->getLeft());
        if(treeNode->getRight() != NULL )
             q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```

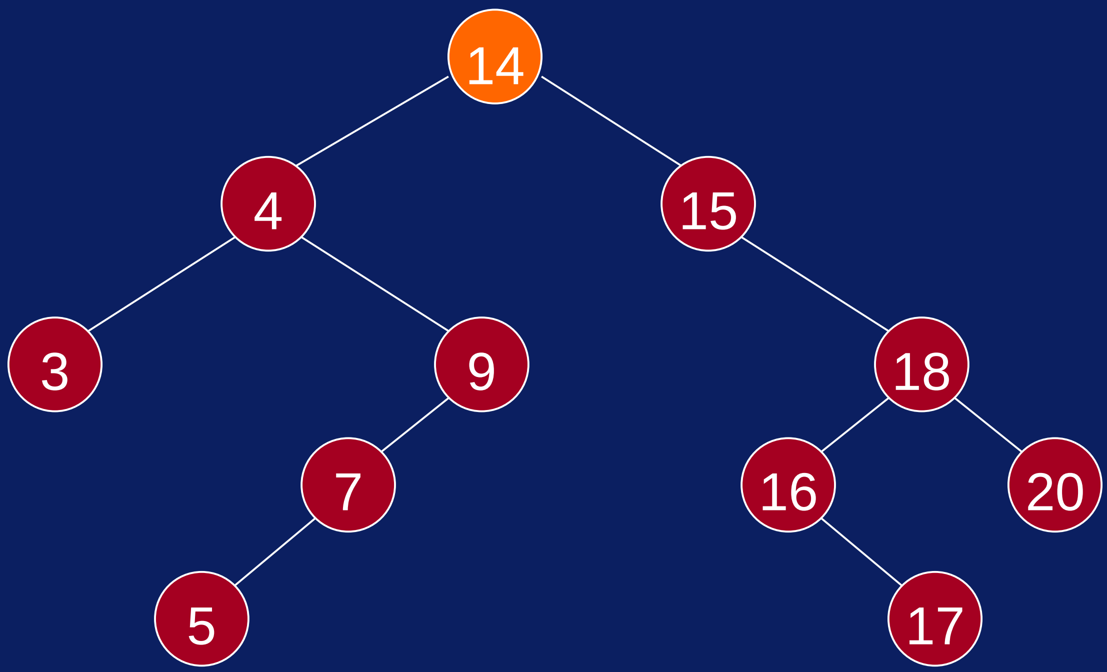
Level-order Traversal

```
void levelorder(TreeNode<int>* treeNode)
{
    Queue<TreeNode<int>* > q;
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
        treeNode = q.dequeue();
        cout << *(treeNode->getInfo()) << " ";
        if(treeNode->getLeft() != NULL )
             q.enqueue( treeNode->getLeft());
        if(treeNode->getRight() != NULL )
             q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```

Level-order Traversal

```
void levelorder(TreeNode<int>* treeNode)
{
    Queue<TreeNode<int>* > q;
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
        treeNode = q.dequeue();
        cout << *(treeNode->getInfo()) << " ";
        if(treeNode->getLeft() != NULL )
             q.enqueue( treeNode->getLeft());
        if(treeNode->getRight() != NULL )
             q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```

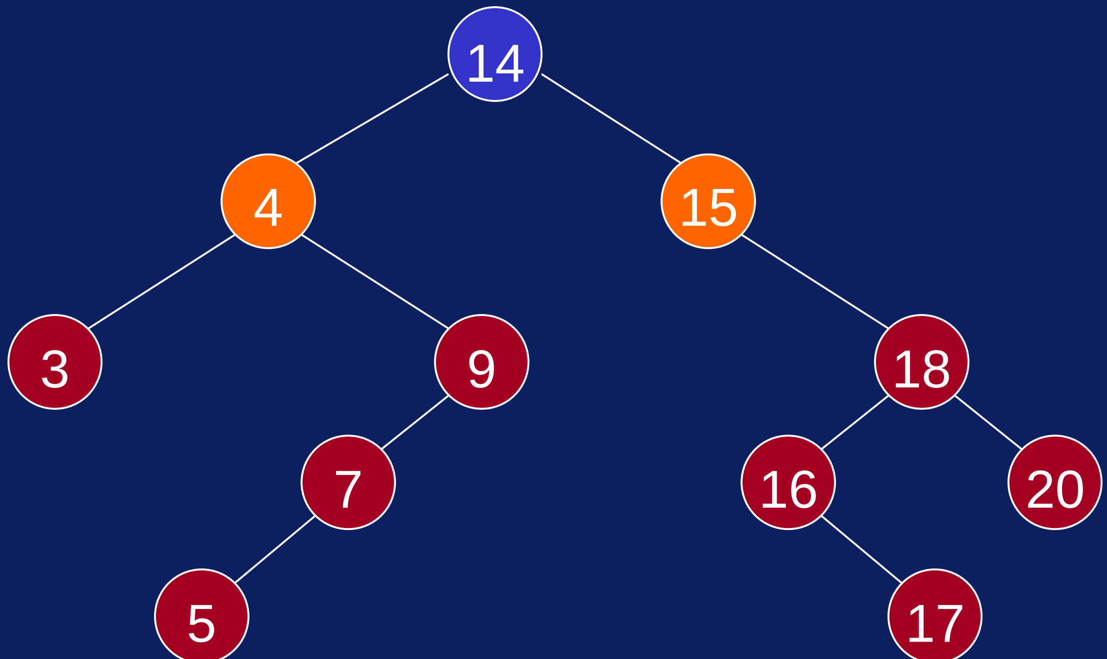
Level-order Traversal



Queue: 14

Output:

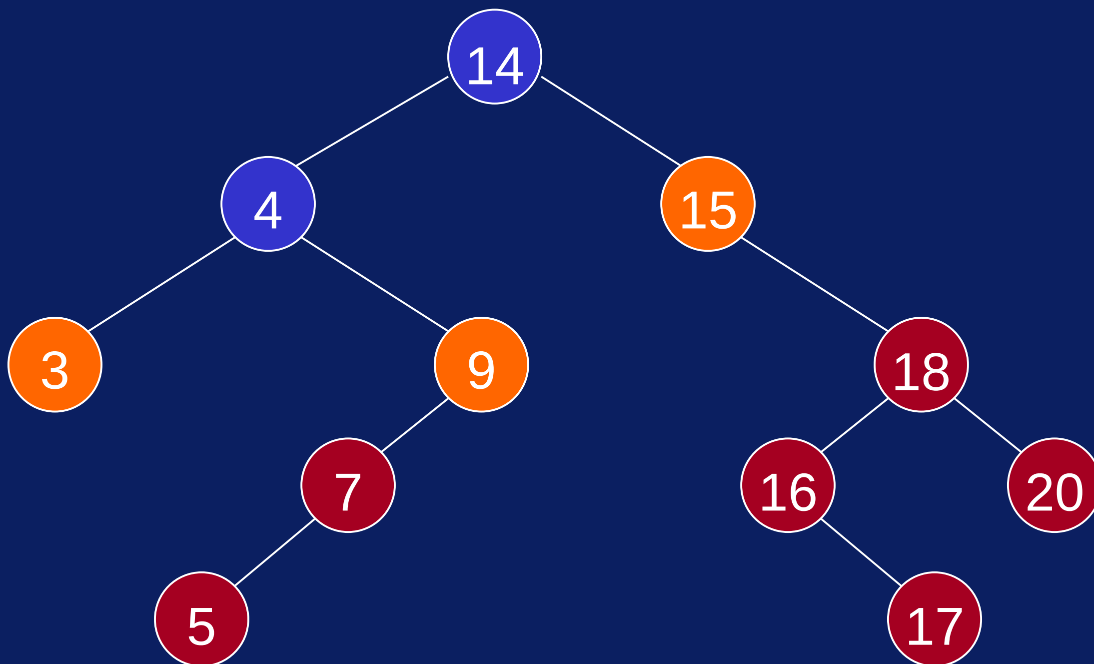
Level-order Traversal



Queue: 4 15

Output: 14

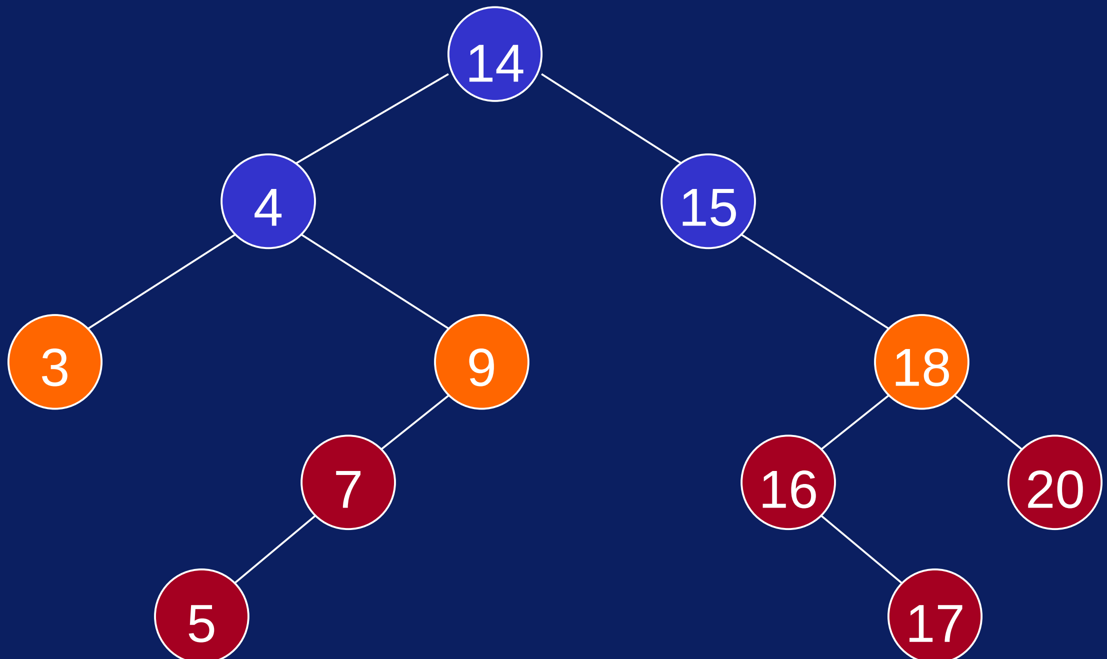
Level-order Traversal



Queue: 15 3 9

Output: 14 4

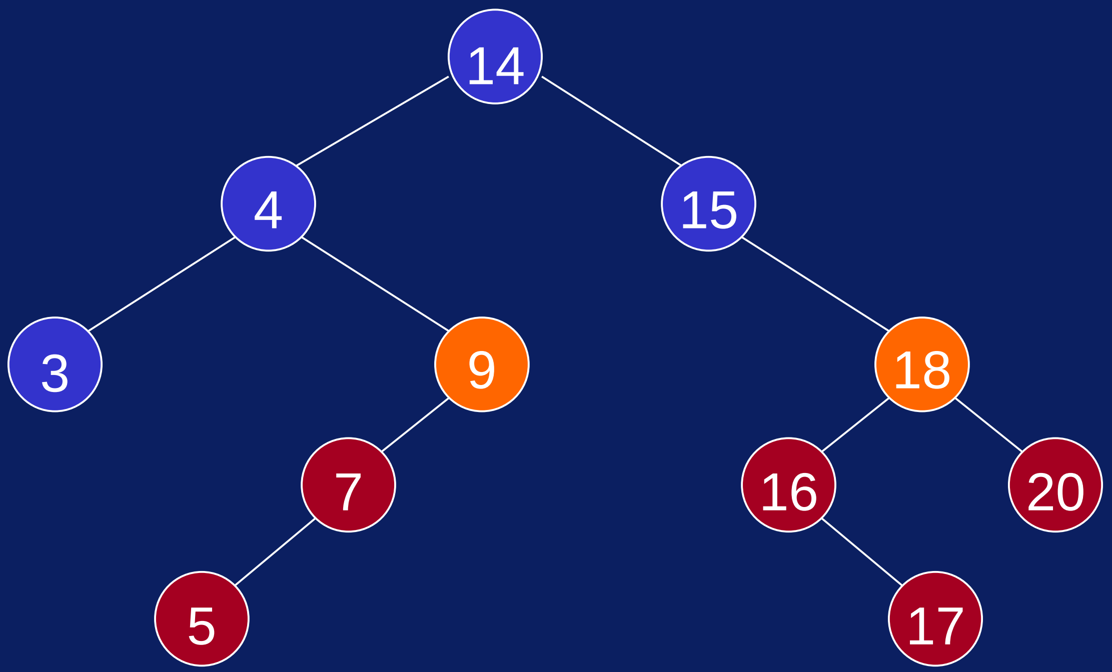
Level-order Traversal



Queue: 3 9 18

Output: 14 4 15

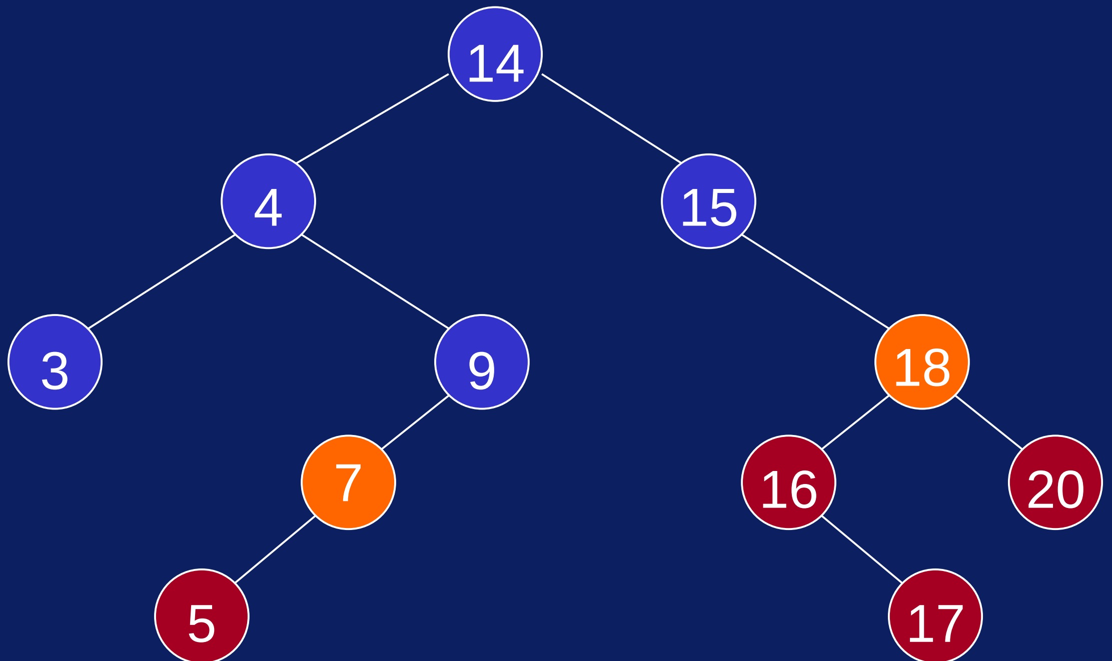
Level-order Traversal



Queue: 9 18

Output: 14 4 15 3

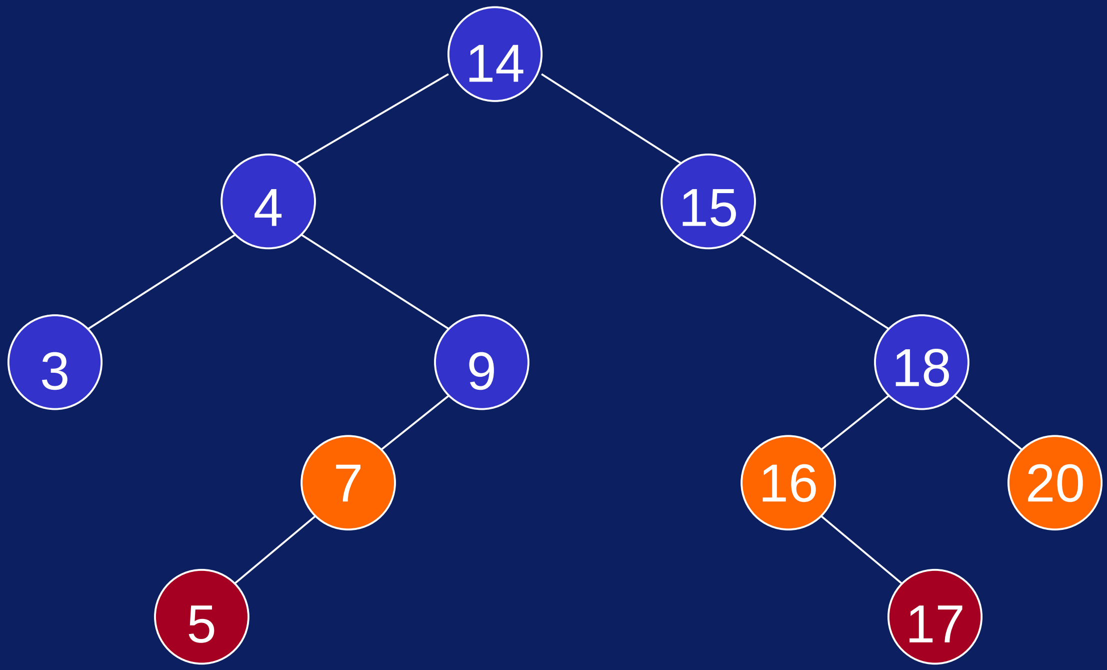
Level-order Traversal



Queue: 18 7

Output: 14 4 15 3 9

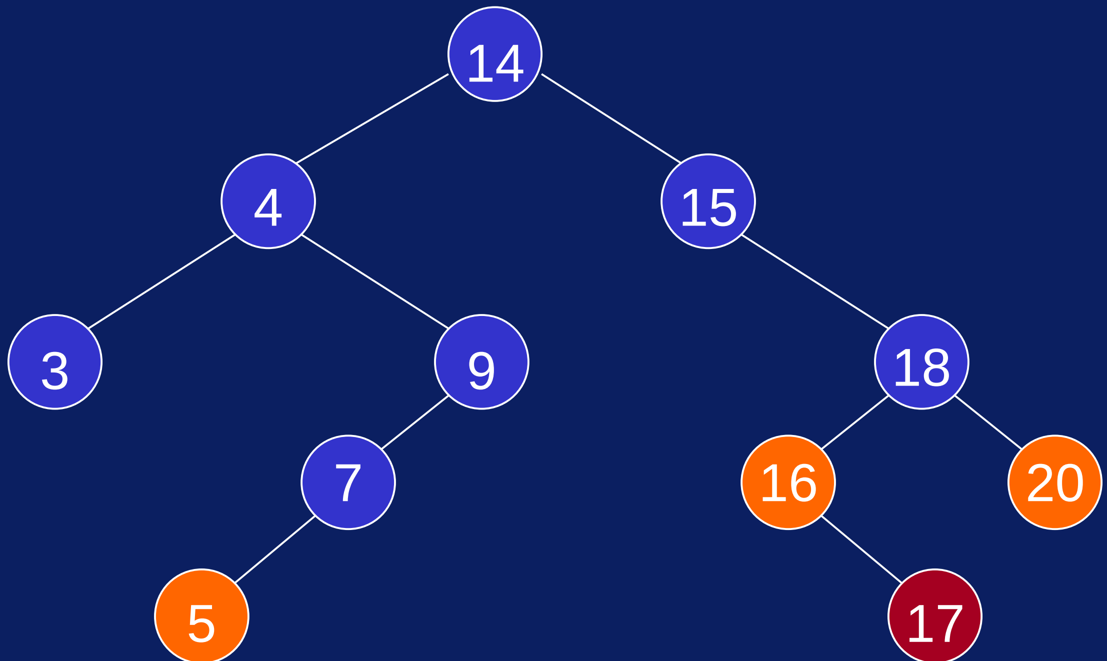
Level-order Traversal



Queue: 7 16 20

Output: 14 4 15 3   9 18

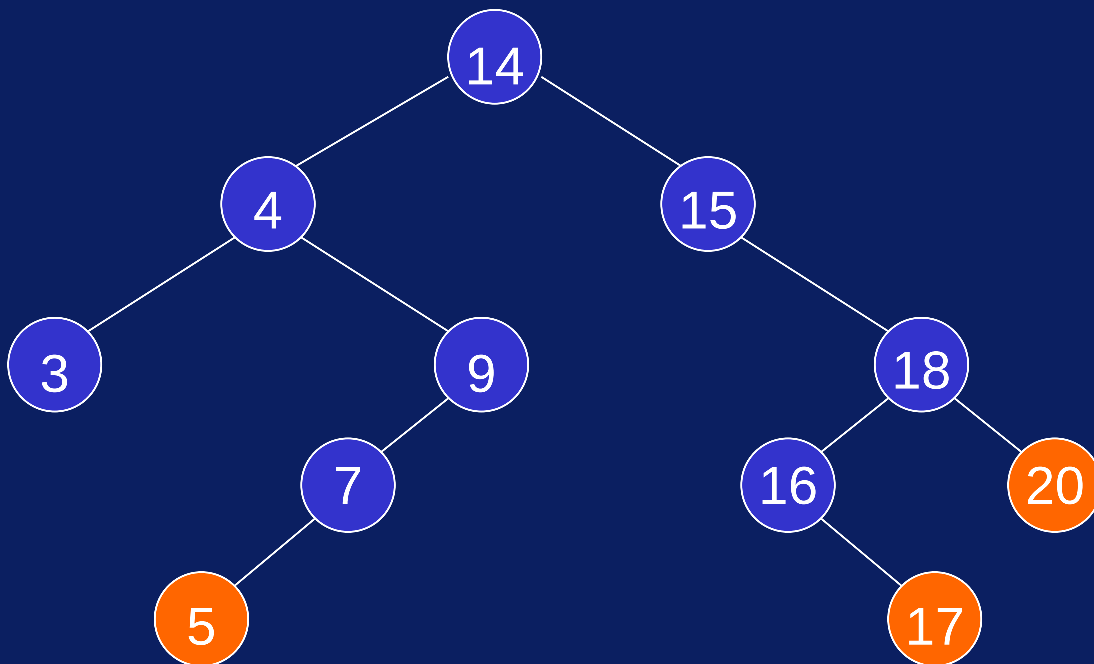
Level-order Traversal



Queue: 16 20 5

Output: 14 4 15 3 9 18 7

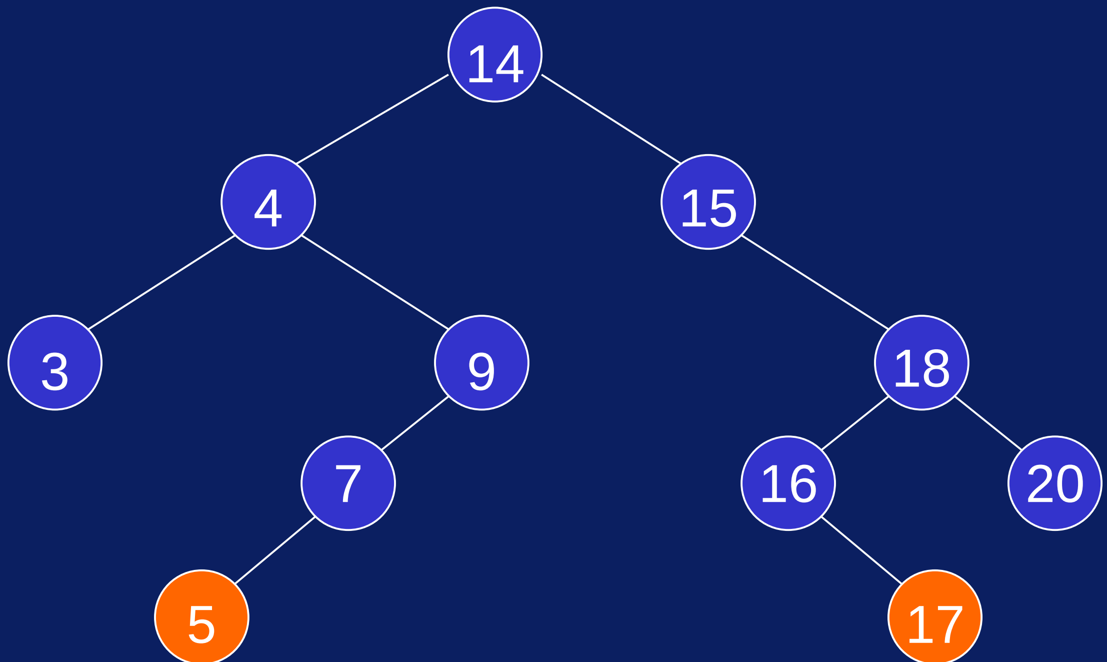
Level-order Traversal



Queue: 20 5 17

Output: 14 4 15 3 9 18 7 16

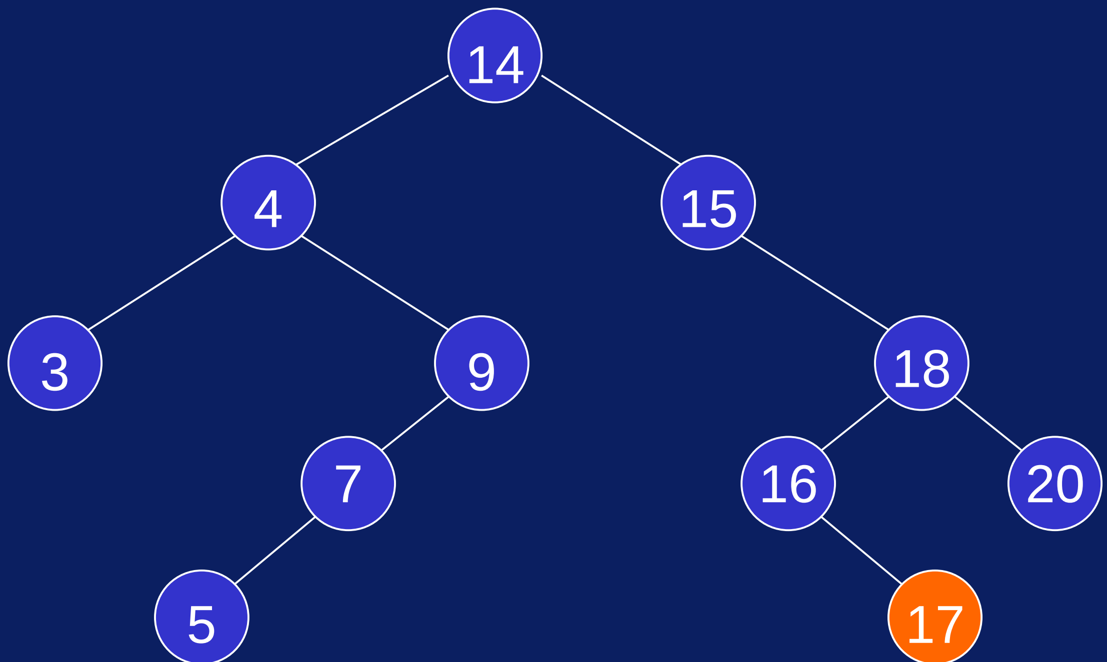
Level-order Traversal



Queue: 5 17

Output: 14 4 15 3   9 18   7 16 20

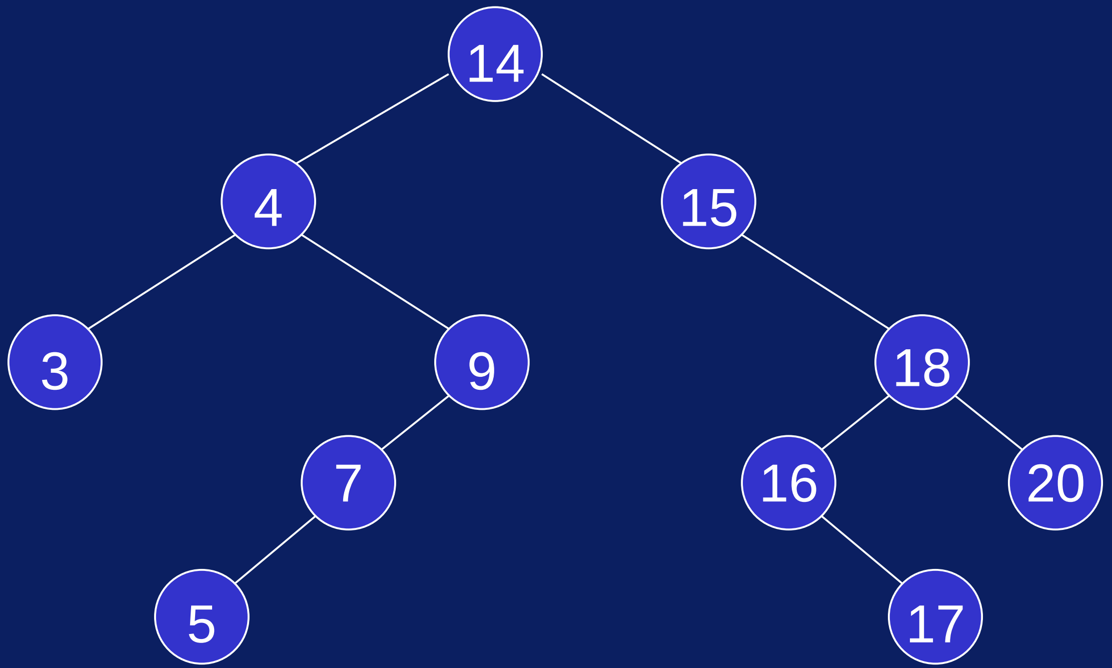
Level-order Traversal



Queue: 17

Output: 14 4 15 3 9 18 7 16 20 5

Level-order Traversal



Queue:

Output: 14 4 15 3 9 18 7 16 20 5 17

Storing other Type of Data

- The examples of binary trees so far have been storing integer data in the tree node.
- This is surely not a requirement. Any type of data can be stored in a tree node.
- Here, for example, is the C++ code to build a tree with character strings.

Binary Search Tree with Strings

```
void wordTree()
{
    TreeNode<char>* root = new TreeNode<char>();
    static char* word[] = "babble", "fable", "jacket",
     "backup", "eagle","daily","gain","bandit","abandon",
     "abash","accuse","economy","adhere","advise","cease",
     "debunk","feeder","genius","fetch","chain", NULL};
    root->setInfo( word[0] );

    for(i=1; word[i]; i++ )
        insert(root, word[i] );
    inorder( root ); cout << endl;
}
```

Binary Search Tree with Strings

```
void wordTree()
{
   TreeNode<char>* root = new TreeNode<char>();
   static char* word[] = "babble", "fable", "jacket",
    "backup", "eagle","daily","gain","bandit","abandon",
    "abash","accuse","economy","adhere","advise","cease",
    "debunk","feeder","genius","fetch","chain", NULL};
   root->setInfo( word[0] );

   for(i=1; word[i]; i++ )
       insert(root, word[i] );
   inorder( root ); cout << endl;
}
```

Binary Search Tree with Strings

```
void wordTree()
{
   TreeNode<char>* root = new TreeNode<char>();
   static char* word[] = "babble", "fable", "jacket",
    "backup", "eagle","daily","gain","bandit","abandon",
    "abash","accuse","economy","adhere","advise","cease",
    "debunk","feeder","genius","fetch","chain", NULL};
   root->setInfo( word[0] );

   for(i=1; word[i]; i++ )
       insert(root, word[i] );
   inorder( root ); cout << endl;
}
```

Binary Search Tree with Strings

```
void wordTree()
{
   TreeNode<char>* root = new TreeNode<char>();
   static char* word[] = "babble", "fable", "jacket",
    "backup", "eagle","daily","gain","bandit","abandon",
    "abash","accuse","economy","adhere","advise","cease",
    "debunk","feeder","genius","fetch","chain", NULL};
   root->setInfo( word[0] );

   for(i=1; word[i]; i++ )
       insert(root, word[i] );
   inorder( root ); cout << endl;
}
```

Binary Search Tree with Strings

```
void wordTree()
{
   TreeNode<char>* root = new TreeNode<char>();
   static char* word[] = "babble", "fable", "jacket",
    "backup", "eagle","daily","gain","bandit","abandon",
    "abash","accuse","economy","adhere","advise","cease",
    "debunk","feeder","genius","fetch","chain", NULL};
   root->setInfo( word[0] );

   for(i=1; word[i]; i++ )
       insert(root, word[i] );
   inorder( root ); cout << endl;
}
```

Binary Search Tree with Strings

```
void insert(TreeNode<char>* root, char* info)
{
    TreeNode<char>* node = new TreeNode<char>(info);
    TreeNode<char> *p, *q;
    p = q = root;
    while( strcmp(info, p->getInfo()) != 0 && q != NULL )
    {
        p = q;
        if( strcmp(info, p->getInfo()) < 0 )
            q = p->getLeft();
        else
            q = p->getRight();
    }
```

Binary Search Tree with Strings

```
void insert(TreeNode<char>* root, char* info)
{
    TreeNode<char>* node = new TreeNode<char>(info);
    TreeNode<char> *p, *q;
    p = q = root;
    while( strcmp(info, p->getInfo()) != 0 && q != NULL )
    {
        p = q;
        if( strcmp(info, p->getInfo()) < 0 )
            q = p->getLeft();
        else
            q = p->getRight();
    }
```

Binary Search Tree with Strings

```
void insert(TreeNode<char>* root, char* info)
{
    TreeNode<char>* node = new TreeNode<char>(info);
    TreeNode<char> *p, *q;
    p = q = root;
    while( strcmp(info, p->getInfo()) != 0 && q != NULL )
    {
        p = q;
        if( strcmp(info, p->getInfo()) < 0 )
            q = p->getLeft();
        else
            q = p->getRight();
    }
```

Binary Search Tree with Strings

```
void insert(TreeNode<char>* root, char* info)
{
    TreeNode<char>* node = new TreeNode<char>(info);
    TreeNode<char> *p, *q;
    p = q = root;
    while( strcmp(info, p->getInfo()) != 0 && q != NULL )
    {
        p = q;
        if( strcmp(info, p->getInfo()) < 0 )
            q = p->getLeft();
        else
            q = p->getRight();
    }
```

Binary Search Tree with Strings

```
void insert(TreeNode<char>* root, char* info)
{
    TreeNode<char>* node = new TreeNode<char>(info);
    TreeNode<char> *p, *q;
    p = q = root;
    while( strcmp(info, p->getInfo()) != 0 && q != NULL )
    {
        p = q;
        if( strcmp(info, p->getInfo()) < 0 )
            q = p->getLeft();
        else
            q = p->getRight();
    }
```

Binary Search Tree with Strings

```
   if( strcmp(info, p->getInfo()) == 0 ){
      cout << "attempt to insert duplicate: " << *info
            << endl;
      delete node;
   }
   else if( strcmp(info, p->getInfo()) < 0 )
      p->setLeft( node );
   else
      p->setRight( node );
}
```

Binary Search Tree with Strings

```
    if( strcmp(info, p->getInfo()) == 0 ){
        cout << "attempt to insert duplicate: " << *info
                << endl;
        delete node;
    }
    else if( strcmp(info, p->getInfo()) < 0 )
        p->setLeft( node );
    else
        p->setRight( node );
}
```

Binary Search Tree with Strings

```
    if( strcmp(info, p->getInfo()) == 0 ){
        cout << "attempt to insert duplicate: " << *info
                << endl;
        delete node;
    }
    else if( strcmp(info, p->getInfo()) < 0 )
        p->setLeft( node );
    else
        p->setRight( node );
}
```

Binary Search Tree with Strings

```
Output:
    abandon
    abash
    accuse
    adhere
    advise
    babble
    backup
    bandit
    cease
    chain
    daily
    debunk
    eagle
    economy
    fable
    feeder
    fetch
    gain
    genius
    jacket
```

Binary Search Tree with Strings

```
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket
```

- Notice that the words are sorted in increasing order when we traversed the tree in inorder manner.

Binary Search Tree with Strings

```
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket
```

- Notice that the words are sorted in increasing order when we traversed the tree in inorder manner.
- This should not come as a surprise if you consider how we built the BST.

Binary Search Tree with Strings

```
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket
```

- Notice that the words are sorted in increasing order when we traversed the tree in inorder manner.
- This should not come as a surprise if you consider how we built the BST.
- For a given node, values less than the info in the node were all in the left subtree and values greater or equal were in the right.

Binary Search Tree with Strings

```
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket
```

- Notice that the words are sorted in increasing order when we traversed the tree in inorder manner.
- This should not come as a surprise if you consider how we built the BST.
- For a given node, values less than the info in the node were all in the left subtree and values greater or equal were in the right.
- Inorder prints the left subtree, then the node finally the right subtree.

Binary Search Tree with Strings

```
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket
```

- Notice that the words are sorted in increasing order when we traversed the tree in inorder manner.
- This should not come as a surprise if you consider how we built the BST.
- For a given node, values less than the info in the node were all in the left subtree and values greater or equal were in the right.
- Inorder prints the left subtree, then the node finally the right subtree.
- Building a BST and doing an inorder traversal leads to a sorting algorithm.

Binary Search Tree with Strings

```
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket
```

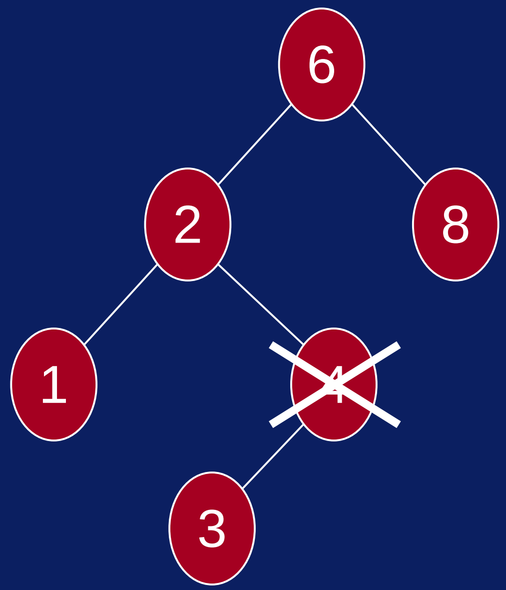
- Notice that the words are sorted in increasing order when we traversed the tree in inorder manner.
- This should not come as a surprise if you consider how we built the BST.
- For a given node, values less than the info in the node were all in the left subtree and values greater or equal were in the right.
- Inorder prints the left subtree, then the node finally the right subtree.
- Building a BST and doing an inorder traversal leads to a sorting algorithm.

Deleting a node in BST

- As is common with many data structures, the hardest operation is deletion.
- Once we have found the node to be deleted, we need to consider several possibilities.
- If the node is a *leaf*, it can be deleted immediately.

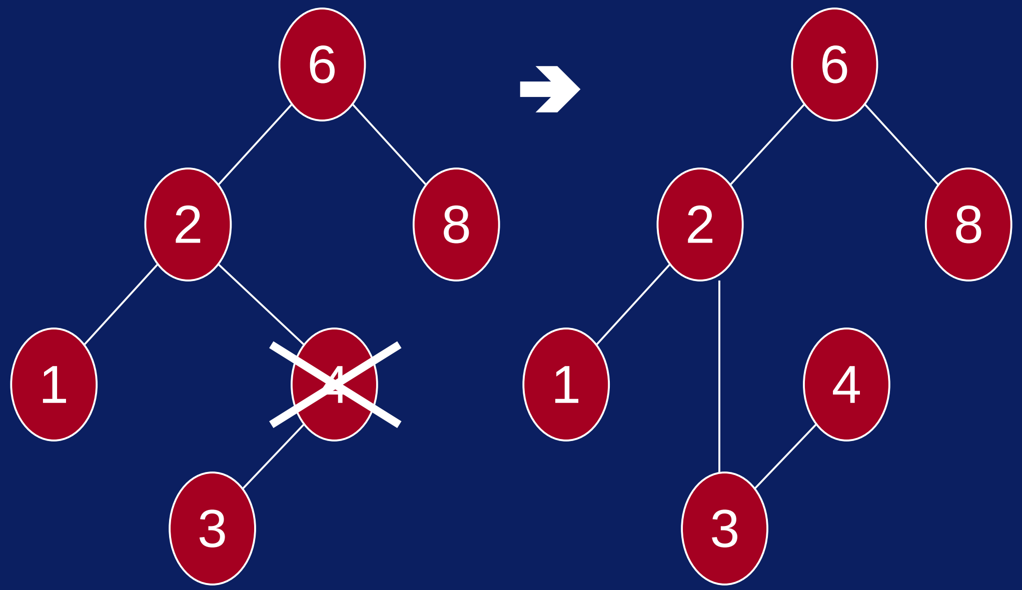
Deleting a node in BST

- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to inorder successor.



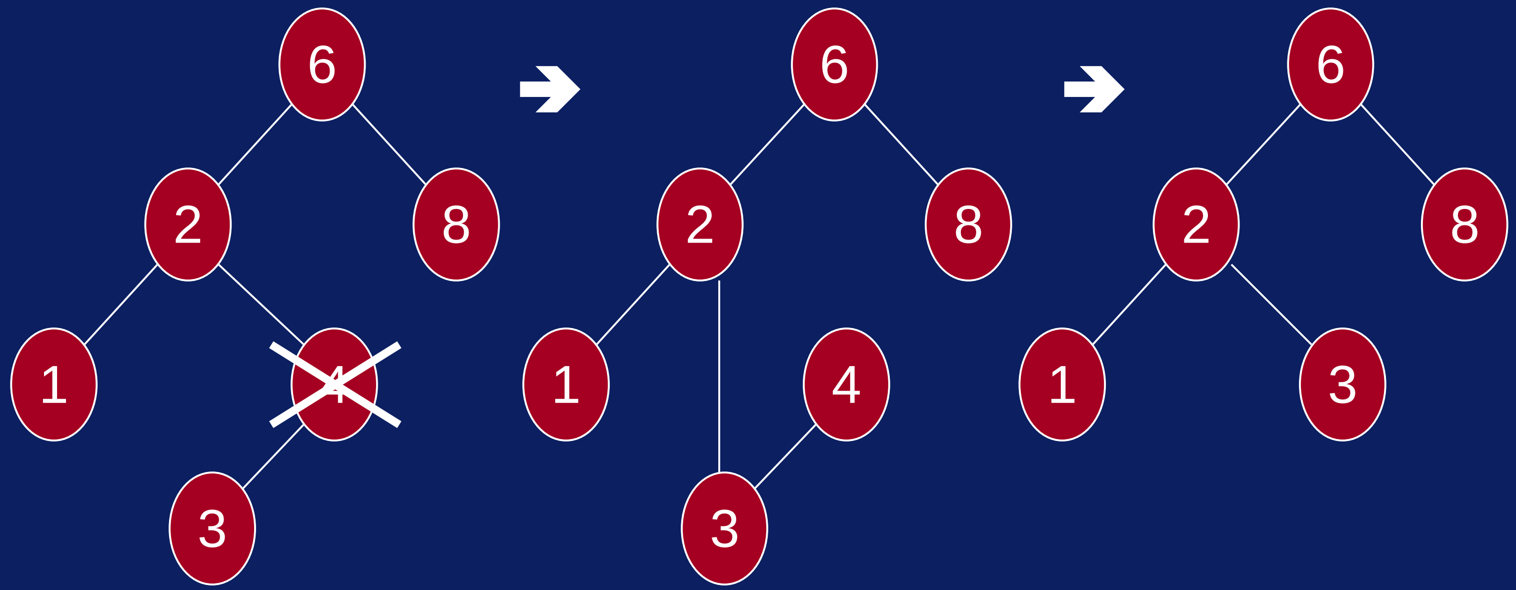
Deleting a node in BST

- The inorder traversal order has to be maintained after the delete.



Deleting a node in BST

- The inorder traversal order has to be maintained after the delete.

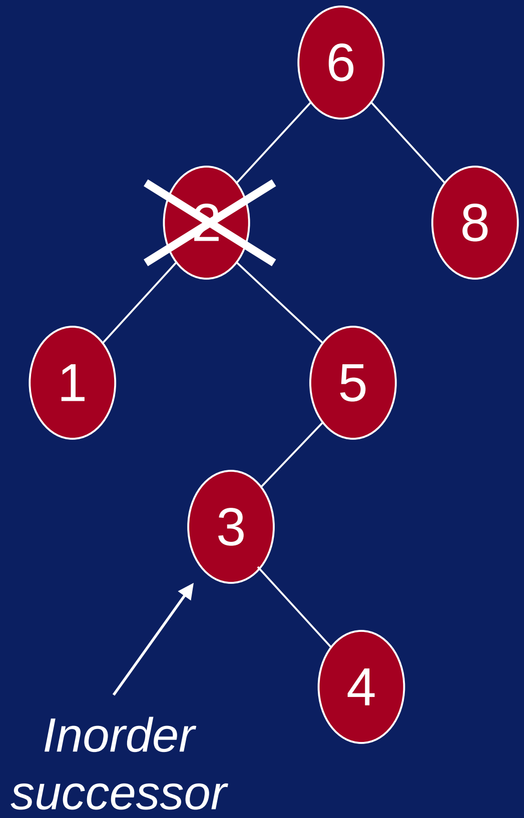


Deleting a node in BST

- The complicated case is when the node to be deleted has both left and right subtrees.
- The strategy is to replace the data of this node with the smallest data of the right subtree and recursively delete that node.

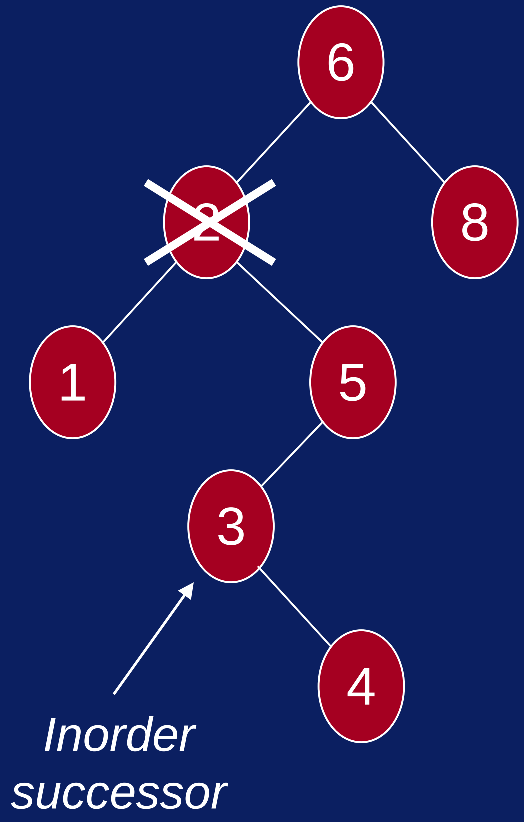
Deleting a node in BST

Delete(2): locate inorder successor



Deleting a node in BST

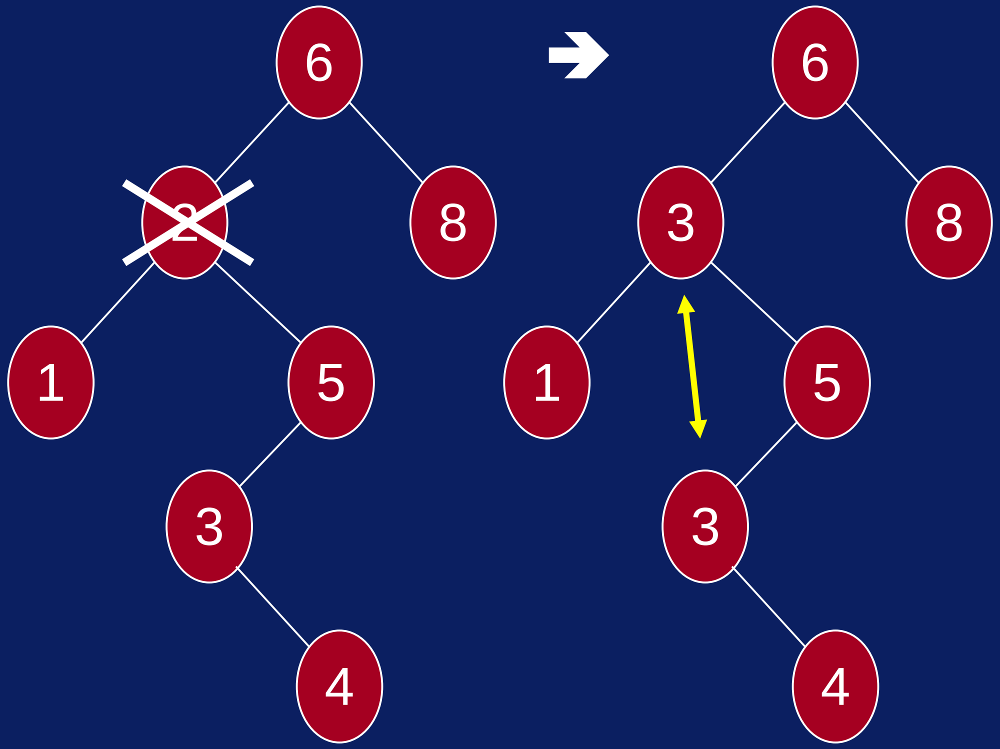
Delete(2): locate inorder successor



- Inorder successor will be the left-most node in the right subtree of 2.
- The inorder successor will not have a left child because if it did, that child would be the left-most node.

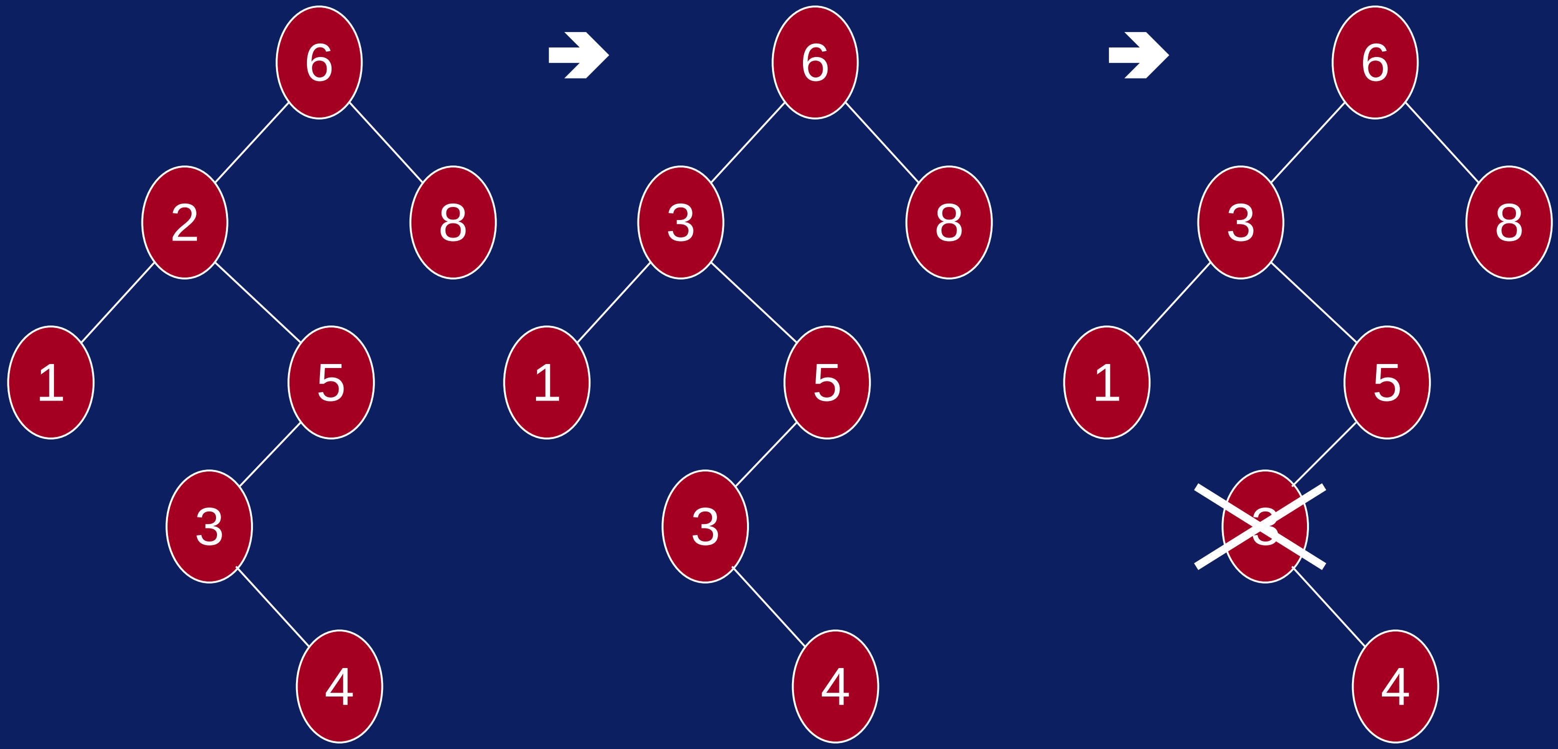
Deleting a node in BST

Delete(2): copy data from inorder successor



Deleting a node in BST

Delete(2): remove the inorder successor



Deleting a node in BST

Delete(2)