Chapter 5 Trees

5.1 Introduction

5.1.1 Terminology

Definition: A tree is a finite set of one or more nodes such that

(1) There is a specially designated node called root.

(2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. $T_1, \ldots, T_n$ are called subtrees of the root.

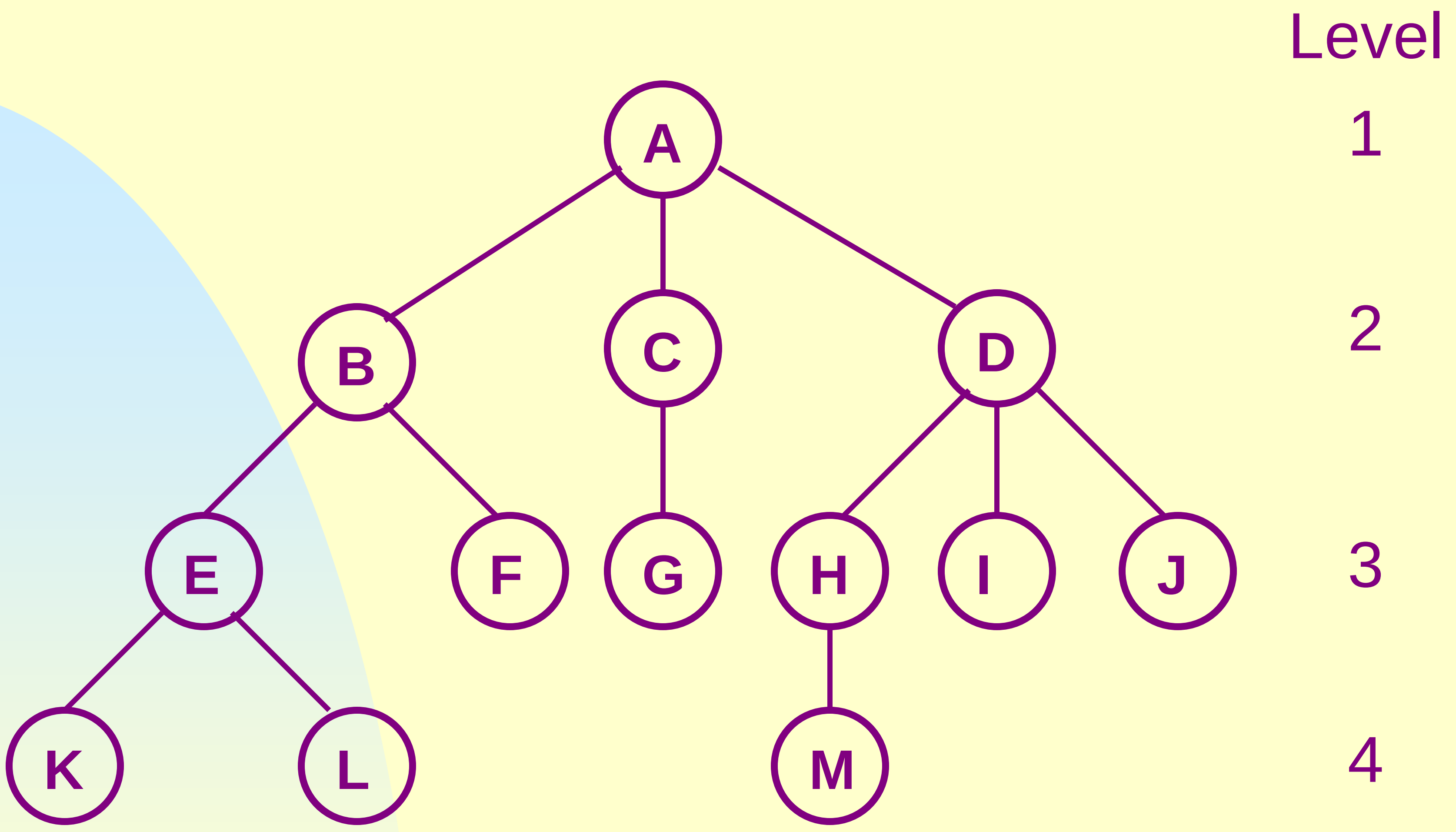


Fig. 5.2 A sample tree

A **node** stands for the item of information plus the branches to other nodes.

The number of subtrees of a node is its **degree**. Nodes with degree 0 are **leaf** or **terminal** nodes, others are **nonterminals**.

The roots of the subtrees of a node X are the **children** of X, X is the **parent** of its children. Children of the same parent are **siblings**.

The **degree of a tree** is the maximum of the degree of the nodes in the tree.

The ancestors of a node are all the nodes along the path from the root to that node.

The level of a node is defined by letting the root be at level 1, if a node is at level l, then its children are at l+1.

The height or depth of a tree is the maximum level of any node in the tree.

5.1.2 Representation of Trees

1. Represented by Generalized lists.

2. For a tree of degree k, we could use a tree node that has fields for data and k pointers to the children as below:

Data	Child1	Child2	…	Child k

Fig.5.4 Possible node structure for a tree of degree k

However, this is very wasteful of space as Lemma5.1 in the next slide shows.

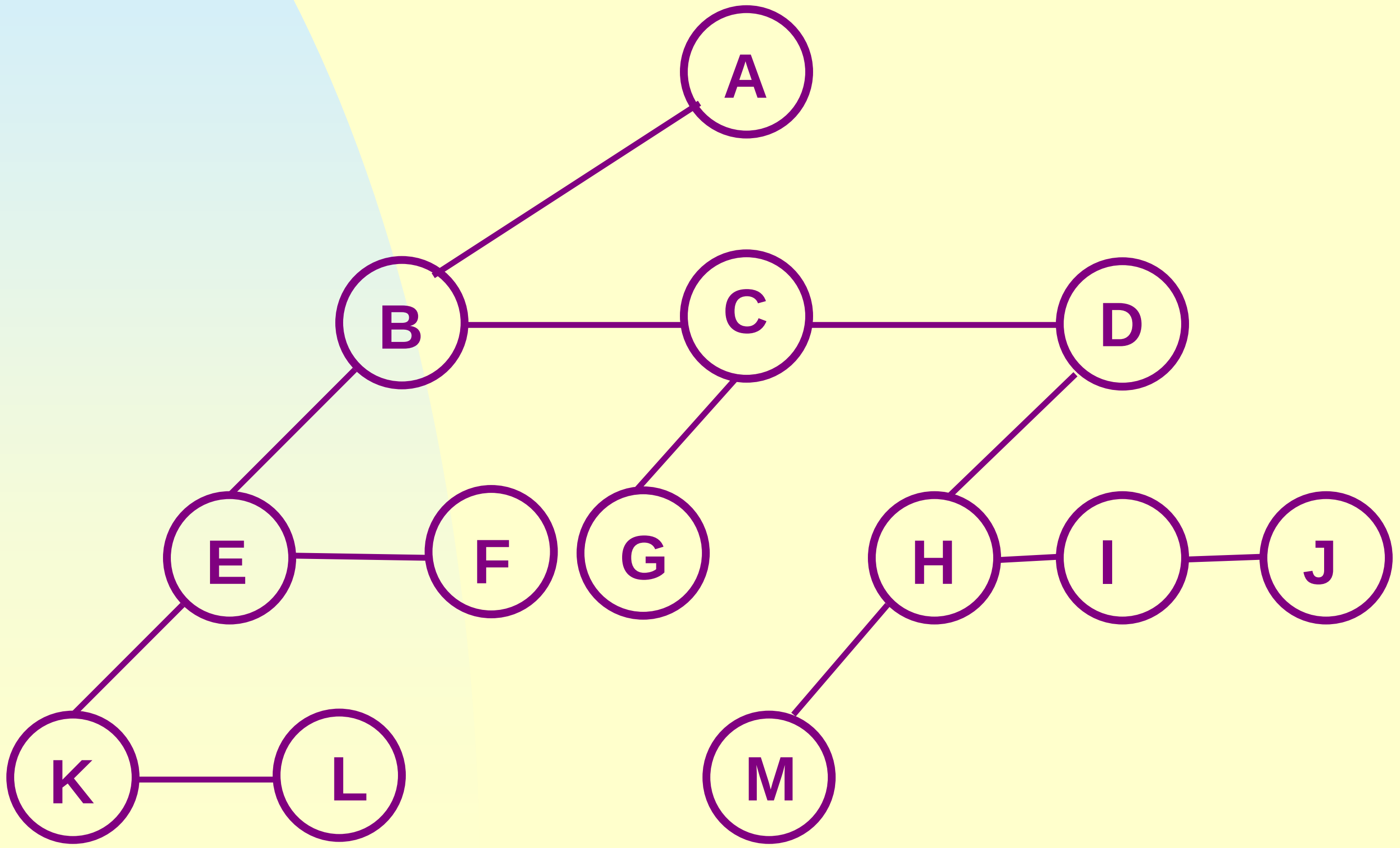
Lemma5.1: If T is a k-ary tree with n nodes, each having a fixed size as in Fig.5.4, then n(k-1)+1 of the nk child fields are 0, $n \geq 1$.

Proof:

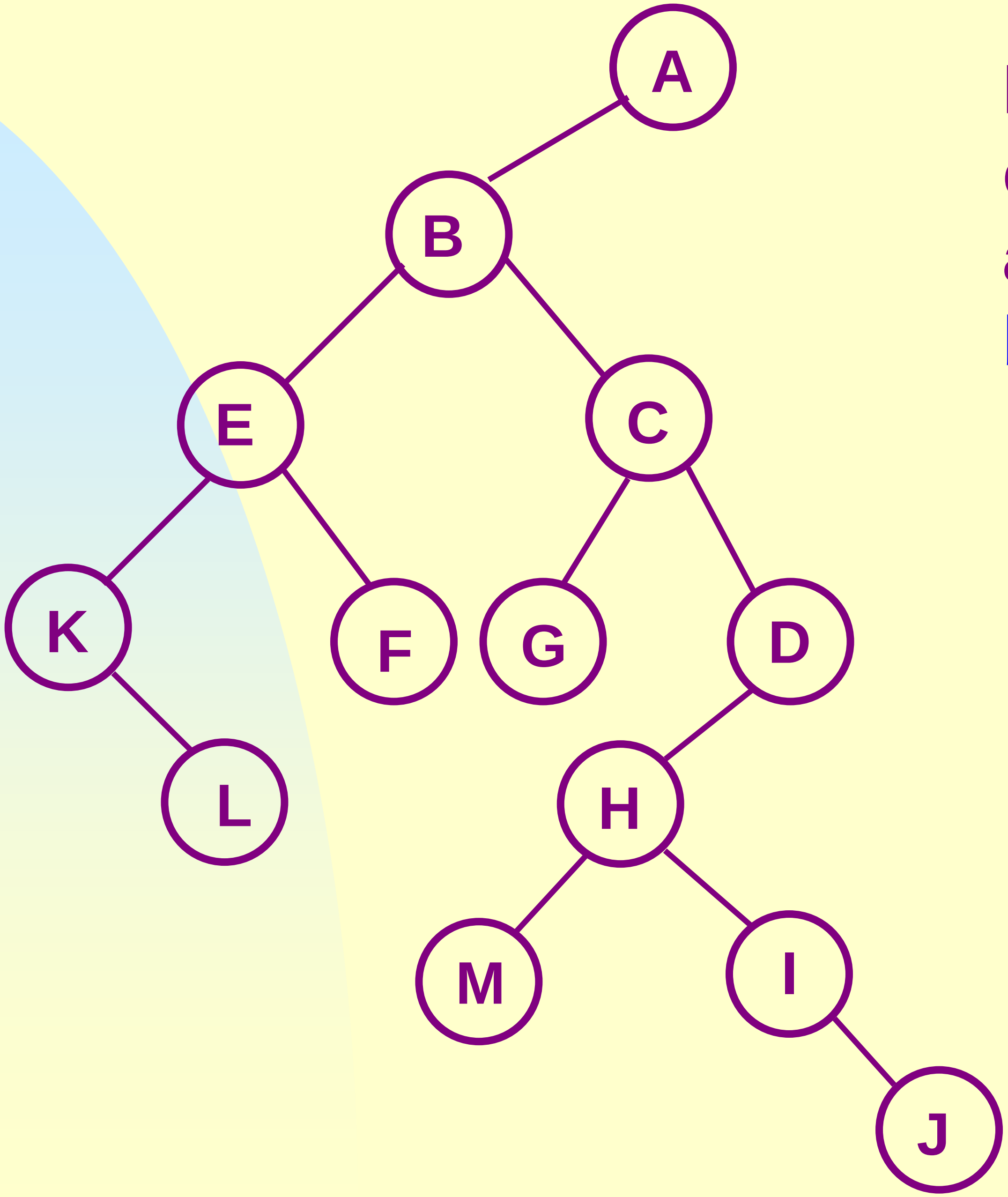
- **each non-zero child field points to a node**
- **there is exactly one pointer to each node other than the root**
- **the number of non-zero child fields in an n node tree is n-1**
- **the number of zero fields is nk-(n-1)=n(k-1)+1.**

3. Left child-right sibling representation

data	
left child	right sibling



4. Left child-right child representation



Left child-right child trees are also knows as binary trees

5.2 Binary Trees

5.2.1 The Abstract Data Type

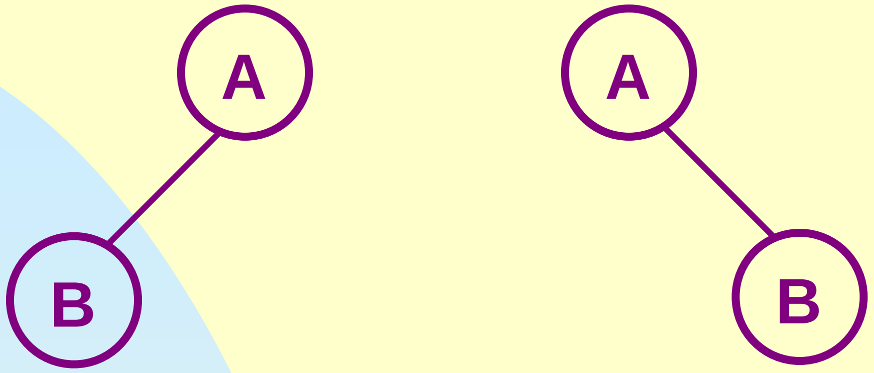
Definition: A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

ADT 5.1

```
template <class T>
class BinaryTree
{ // A finite set of nodes either empty or consisting of
  // a root node, left BinaryTree and right BinaryTree.
public:
    BinaryTree ();
    // creates an empty binary tree

    bool IsEmpty ();
    // return true iff the binary tree is empty

    BinaryTree(BinaryTree<T>& bt1, T& item,
                                        BinaryTree<T>& bt2);
    // creates a binary tree whose left subtree is bt1,
    // right subtree is bt2, and root node contain item.

    BinaryTree  LeftSubtree();
    // return the left subtree of *this
    T RootData();
    // return the data in the root of *this

    BinaryTree  RightSutree();
    // ireturn the right subtree of *this
};
```

Distinction between a binary tree and a tree:

- **No tree has 0 node, but there is an empty binary tree.**
- **In a binary tree, we distinguish between the order of the child; in a tree, we do not.**
- **A nonempty binary tree's root always has two subtrees: the left and the right, although they may be empty.**



So the above two binary trees are different: the first has an empty right subtree while the second has an empty left subtree. But as trees, they are the same.

In a binary tree, the number of non-empty subtrees of a node is its degree.

5.2.2 Properties of Binary Trees

Lemma 5.2 [Maximum number of nodes]:

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

(2) The maximum number of nodes in a binary tree of depth k is 2^k-1, $k \geq 1$.

Proof:

Let M_i be the maximum number of nodes on level i

(1) Induction on level i.

i=1, only the root, $M_i = 2^{1-1} = 2^0 = 1$.

Assume $M_{i-1} = 2^{(i-1)-1}$ for i>1.

Since each node has a maximum degree of 2, $M_i = 2 * M_{i-1} = 2*2^{i-2} = 2^{i-1}$.

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k} M_i = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1.$$

Lemma 5.3 [Relation between number of leaf nodes and degree-2 nodes]:

For any nonempty binary tree T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof:

Let n_1 be the number of nodes of degree 1 and n the total number of nodes, we have

$$n = n_0 + n_1 + n_2 \qquad (5.1)$$

Each node except for the root has a branch leading into it. If B is the number of branches, then n = B+1. And also $B = n_1 + 2n_2$, hence

$$n = n_1 + 2n_2 + 1 \qquad (5.2)$$

(5.1) – (5.2): $0 = n_0 - n_2 - 1$, i.e., $n_0 = n_2 + 1$.

Definition: A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes, $k \geq 0$.

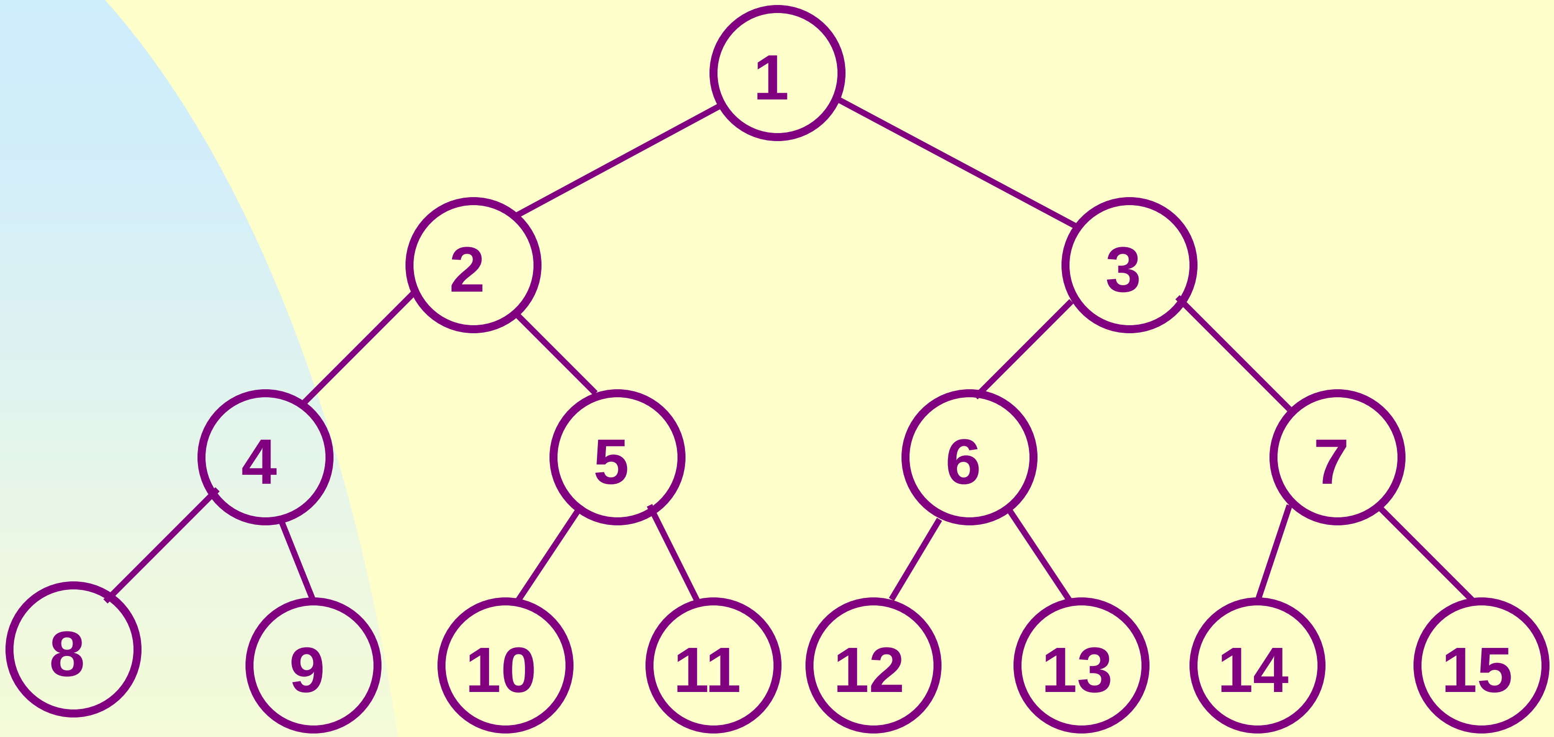


Fig. 5.11 Full binary tree of depth 4

Suppose we number the nodes in a full binary tree starting with the root on level 1, continuing with the node on level 2, and so on. Nodes on any level are numbered from left to right. This **sequentially numbering scheme** gives us the definition of a **complete binary tree**.

Definition: a binary tree with n nodes and depth k is **complete** iff its nodes corresponding to the nodes numbered from 1 to n in the full binary tree of depth k.

From Lemma5.2, the height of a complete binary tree with n nodes is $\lceil \log_2(n+1) \rceil$.

5.2.3 Binary Trees Representations

5.2.3.1 Array Representation

The nodes may be stored in a one dimensional array named tree, with the node sequentially numbered i being stored in tree[i].

Lemma 5.4: if a complete binary tree with n nodes is represented sequentially, then for any nodes with index i, $1 \le i \le n$, we have

(1) parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i=1$, i is the root and has no parent.

(2) LeftChild(i) is at 2i, if $2i \leq n$. If $2i>n$, i has no left child.

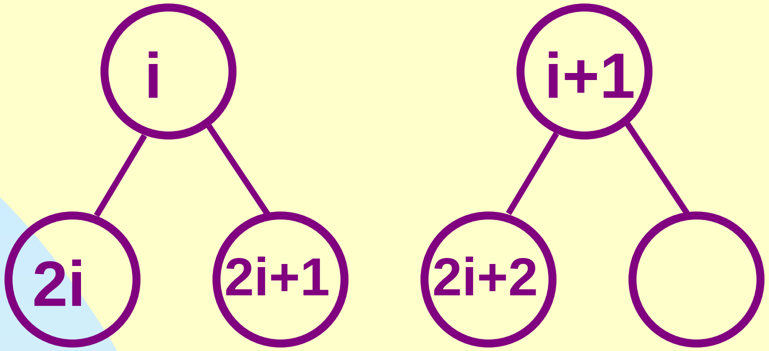
(3) RightChild(i) is at 2i+1, if $2i+1 \leq n$. If $2i+1>n$, i has no right child.

Proof: prove (2). (3) follows from (2) and the nodes numbering from left to right. (1) follows from (2) and (3).

Induction on i.

i=1, clearly the left child is at 2 unless $2>n$, in which case i has no left child.

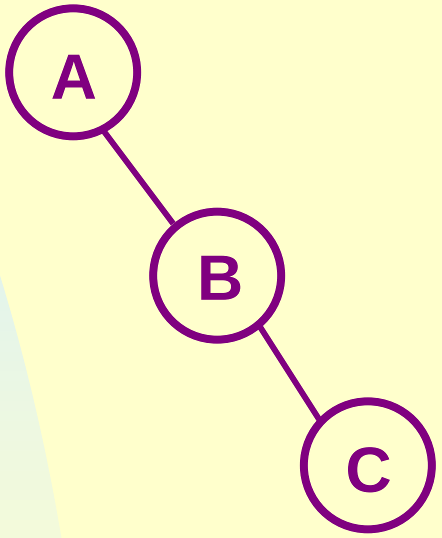
Assume for all j, $1 \leq j \leq i$, LeftChild(j) is at 2j.



The two nodes immediately preceding LeftChild(i+1) are the right and left children of i, hence LeftChild(i+1) is at 2i+2=2(i+1) unless 2(i+1)>n, in which case i+1 has no left child.

The array representation can be used for all binary trees.

- for a complete binary tree---no space wasted.
- for a skewed tree to the right of depth k, in the worst case, it will require 2^k-1 spaces of which only k will be used, as shown below:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
tree	---	A	---	B	---	---	---	C

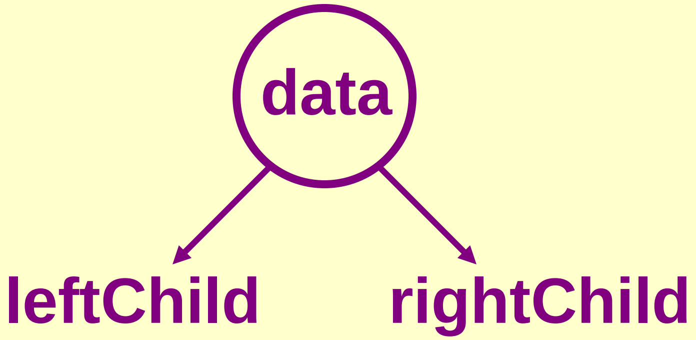
5.2.3.2 Linked Representation

In addtion, the array representation suffers from the general inadequacies of sequential representation: insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes to reflect the change in level number of these nodes.

The problem can be easily overcome through the use of a linked representation.

```
template <class T> class Tree;
class TreeNode {
friend class Tree<T>;
public:
    TreeNode (T& e, TreeNode<T>* left, TreeNode<T>* right)
       {data=e; leftChild=left; rightChild=right;}
private:
    T data;
    TreeNode<T>* leftChild;
    TreeNode<T>* rightChild;
};
```

leftChild	data	rightChild



```
template <class T>
class Tree {
public:
   // Tree operations
…
private:
   TreeNode<T>* root;
};
```

If necessary, a 4th field, parent, may be included in the class TreeNode.

5.3 Binary Tree Traversal and Tree Iterators

5.3.1 Introduction

A full traversal produces a linear order for the nodes in a tree. This linear order may be familiar and useful.

L---moving left, V---visiting the node, R---moving right, and adopt the convention of traversing left before right, we have 3 traversals:

LVR (inorder), LRV (postorder), VLR (preorder).

There is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an expression.

We use the following expression tree to illustrate each of these traversals.

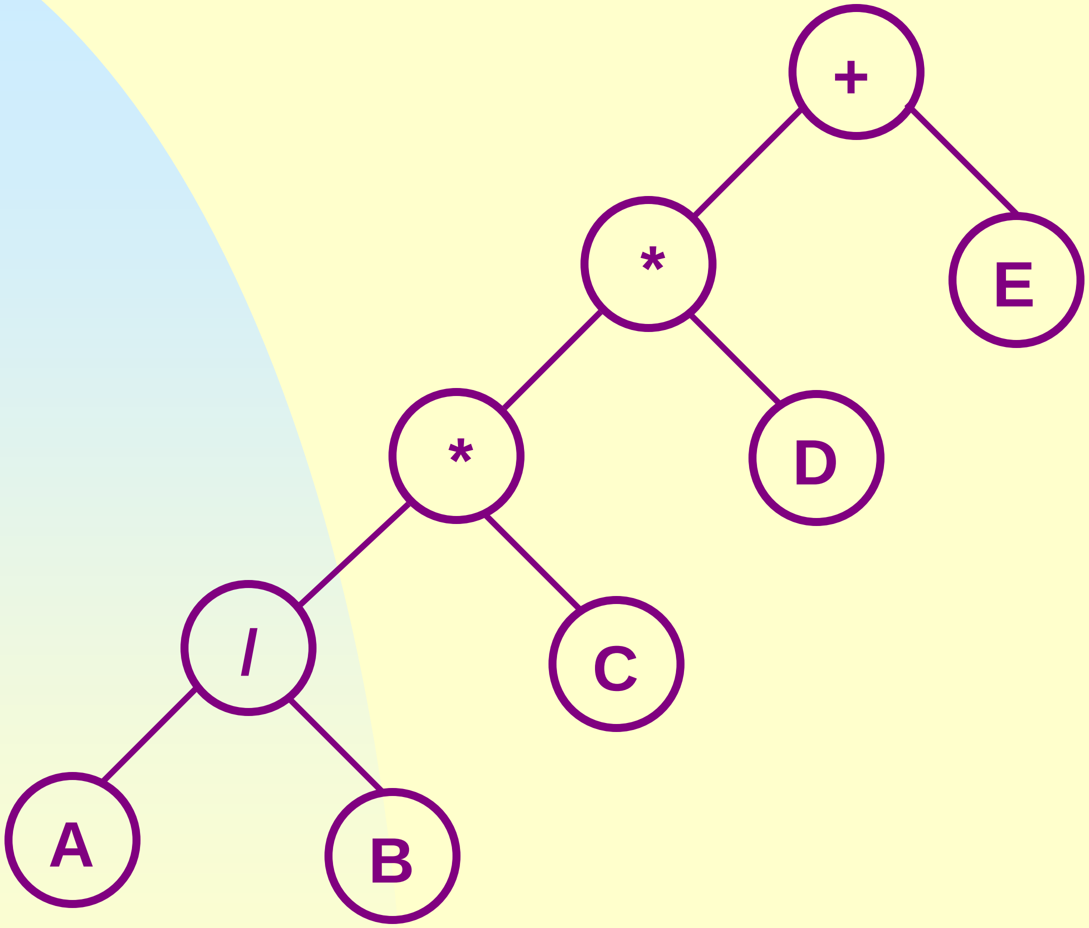


Fig. 5.16 Arithmetic expression tree

5.3.2 Inorder Traversal

```
template <class T>
void Tree<T>::Inorder()
{ // driver as a public member
   Inorder(root);
}

template <class T>
void Tree<T>::Inorder (TreeNode<T>* currentNode)
{ // workhorse as a private member of Tree
    if (CurrentNode) {
        Inorder(currentNode→leftChild);
        Visit(currentNode)
        Inorder(currentNode→rightChild);
     }
}
```

Assume the Visit function has a single line of code:

```
cout <<CurrentNode→data;
```

For the tree of Fig. 5.16, the elements are output as:

A / B * C * D +E (infix)

5.3.3 Preorder Traversal

```
template <class T>
void Tree<T>::Preorder()
{ // Driver.
    Preorder(root);
}
```

```
template <class T>
void Tree<T>::Preorder(TreeNode<T>* currentNode)
{ // workhorse.
     if (currentNode) {
         Visit(currentNode);
         Preorder(currentNode→leftChild);
         Preorder(currentNode→rightChild);
     }
}
```

For the tree of Fig. 5.16, the elements are output as:

+ * * / A B C D E (prefix)

5.3.4 Postorder Traversal

```
template <class T>
void Tree<T>::Postorder()
{ // Driver.
   Postorder(root);
}

template <class T>
void Tree<T>::Postorder (TreeNode<T>* currentNode)
{ // Workhorse.
     if (currentNode) {
        Postorder(currentNode→leftChild);
        Postorder(currentNode→rightChild );
        Visit(currentNode);
     }
}
```

For the tree of Fig. 5.16, the elements are output as:

A B / C * D * E +   (postfix)

5.3.5 Iterative Inorder Traversal

To implement a tree traversal by using iterators, we first need to implement a non-recursive tree traversal algorithm.

A direct way to do so is to use a stack.

```
template <class T>
void Tree<T>::NonrecInorder()
{ // Nonrecursive inorder traversal using a stack
  Stack<TreeNode<T>*> s; // declare and initialize a stack
  TreeNode<T>* currentNode=root;
   while (1) {
      while (currentNode) { // move down leftChild
         s.Push(currentNode); // add to stack
         currentNode=currentNode→leftChild;
      }
      If (s.IsEmpty()) return;
      currentNode=s.Top();
      s.Pop(); // delete from stack
       Visit(currentNode);
       currentNode=currentNode→rightChild;
   }
}
```

The NonrecInorder USES-A template stack.

Definition: A data object of Type A USES-A data object of Type B if a Type A object uses a Type B object to perform a task. Typically, a Type B object is employed in a member function of Type A.

USES-A is similar to IS-IMPLEMENTED-IN-TERMS-OF, but the degree of using the Type B object is less.

Analysis of NonrecInorder:

• n---the number of nodes in the tree.

• every node is placed on the stack once, line 8, 9 and 11 to 15 are executed n times.

• currenetNode will equal 0 once for every 0 link, which is $2n_0 + n_1 = n_0 + n_1 + n_2 + 1 = n + 1$.

The computing time: O(n).

The space required for the stack is equal to the depth of the tree.

Now we use the function NonrecInorder to obtain an inorder iterator for a tree.

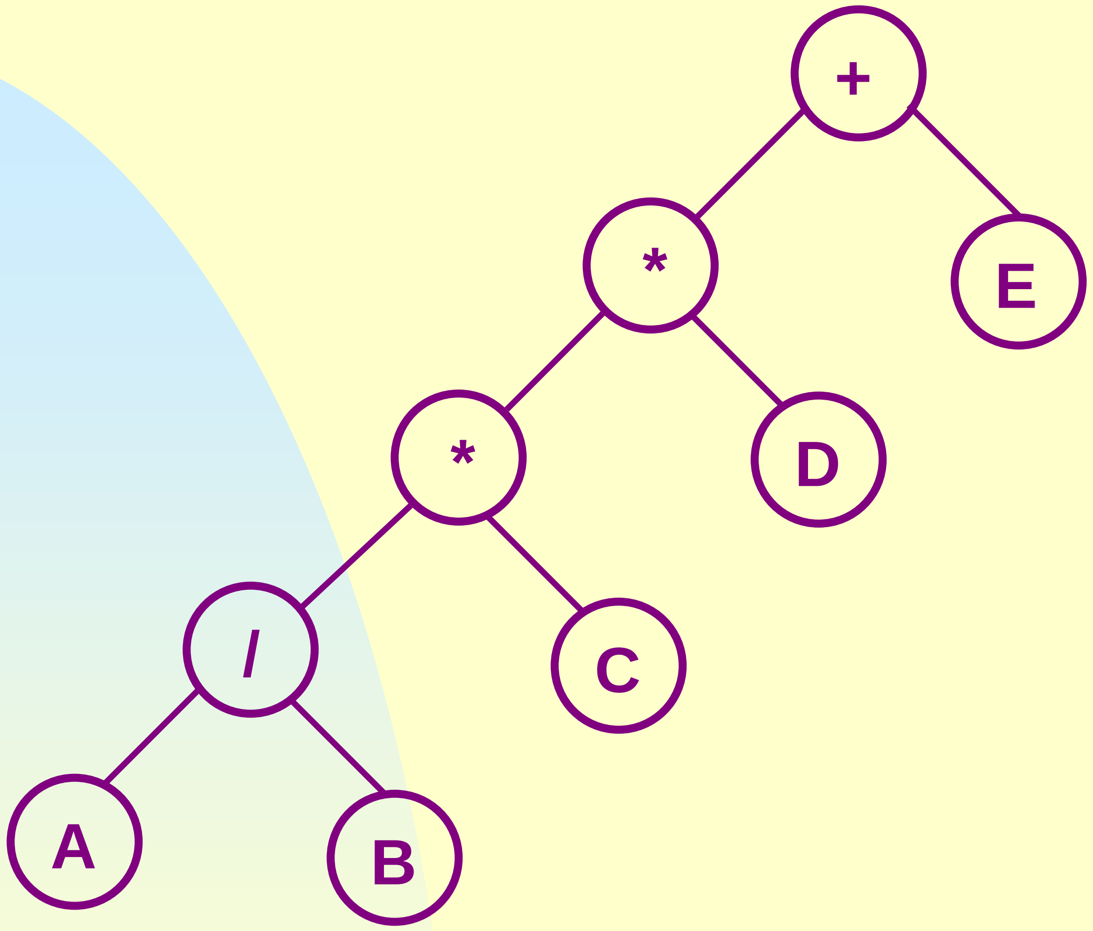
The key observation is that each iteration of the while loop of line 6-16 yields the next element in the inorder traversal of the tree.

```
class InorderIterator { // a public nested member class of Tree
public:
    InorderIterator() {currentNode=root;};
    T* Next( );
private:
    Stack<TreeNode<T>*> s;
    TreeNode<T>* currentNode;
};
```

```
T* InorderIterator::Next()
{
    while (currentNode) {
        s.Push(currentNode);
        currentNode=currentNode→LeftChild;
    }
    if (s.IsEmpty()) return 0;
    currentNode=s.Top();
    T& temp=currentNode→data;
    currentNode=currentNode→rightChild;
    return &temp;
}
```

5.3.6 Level-order Traversal

Level-order traversal visits the nodes in the order suggested in the full binary tree nodes numbering.



The level-order traversal of the left tree is:

+ * E * D / C A B

To implement level-order traversal, we use a queue.

```
template <class T>
void Tree<T>::LevelOrder()
{ // traverse the binary tree in level order
    Queue<TreeNode<T>*> q;
    TreeNode<T>* currentNode=root;
    while (currentNode) {
      Vist(currentNode);
      if (currentNode→leftChild)
            q.Push(currentNode→leftChild);
      if (currentNode→rightChild)
            q.Push(currentNode→rightChild);
      if (q.IsEmpty())return;
      currentNode=q.Front();
      q.Pop();
   }
}
```

Exercises: P267-4, 6

Experiment: P267-10

5.4 Additional Binary Tree Operations

5.4.1 Copying Binary Trees

```
template <class T>
Tree<T>::Tree(const Tree<T>& s)  // driver
{ // Copy constructor
    root = Copy( s.root );
}
```

```
template <class T>
TreeNode<T>* Tree<T>::Copy(TreeNode<T>* origNode)
// workhorse
{ // Return a pointer to an exact copy of the binary
  // tree rooted at origNode
     if (!origNode) return 0;
     return new TreeNode<T>(origNode→data,
                                                  Copy(origNode→leftChild),
                                                  Copy(origNode →rightChild));
}
```

5.4.2 Testing Equality

```
template <class T>
bool Tree<T>::operator==(const Tree& t) const
{
   return Equal(root, t.root);
}

template <class T>
bool Tree<T>::Equal(TreeNode<T>* a, TreeNode<T>* b)
{// Workhorse-
   if ((!a) && (!b)) return true; // both a and b are 0
   return (a && b          // both a and b are non-0
      && (a→data == b→data)                //data is the same
      && Equal(a→leftChild, b→leftChild)          //left equal
      && Equal(a→rightChild, b→rightChild));      //right equal
}
```

Exercises: P272-1, P273-4

5.5 Threaded Binary Trees

5.5.1 Threads

Note that in the linked representation of binary tree, there are n +1 0 links, which is more than actual pointers.

A clever way to make use of these 0 links is to replace them by pointers, called threads, to other nodes in the tree.

The threads are constructed using the following rules:

(1) A 0 rightChild field at node p is replaced by a pointer to the inorder successor of p.

(2) A 0 leftChild field at node p is replaced by a pointer to the inorder predecessor of p.

The following is a threaded tree, in which node E has a predecessor thread pointing to B and a successor thread to A.

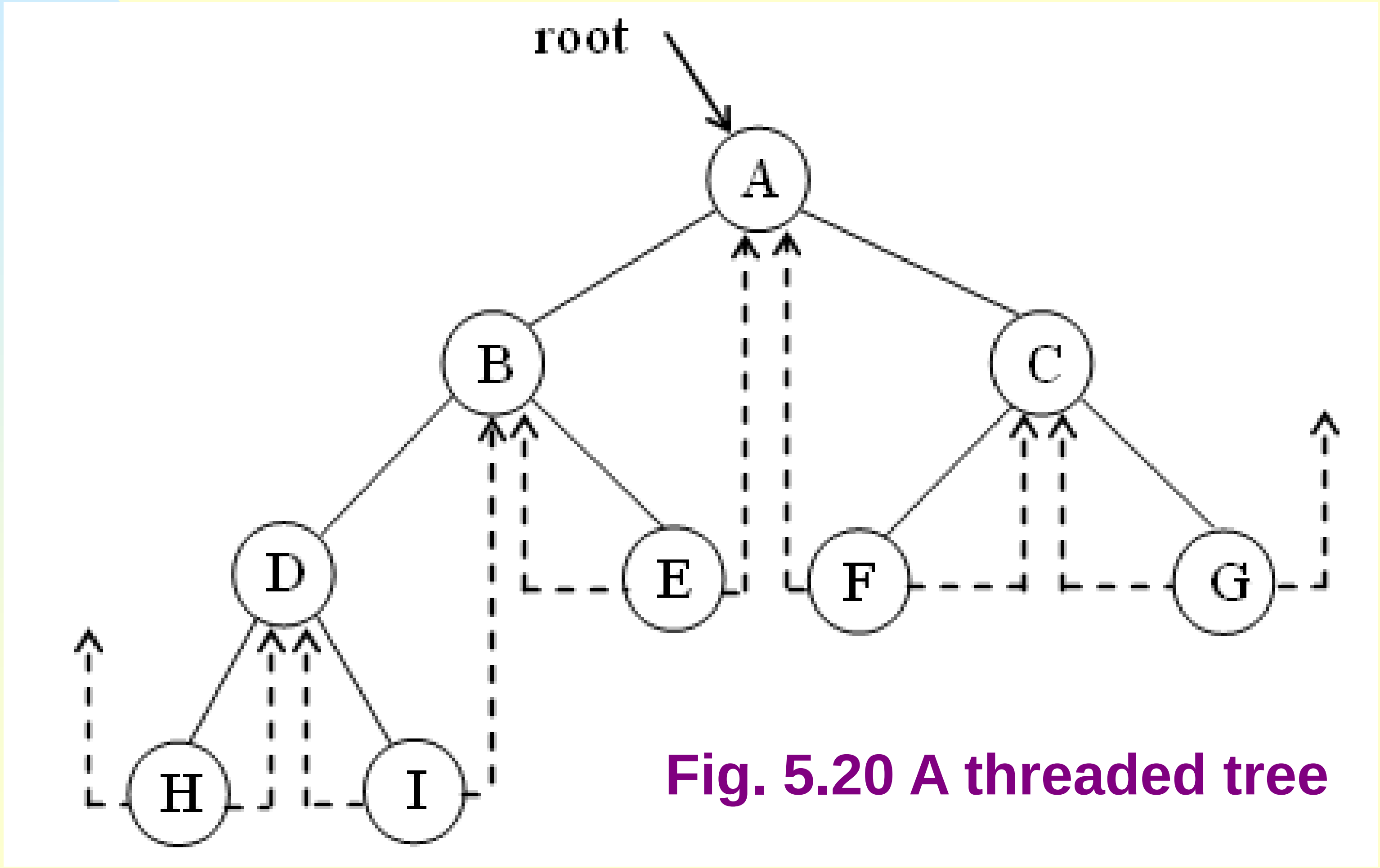


Fig. 5.20 A threaded tree

To distinguish between threads and normal pointers, add two bool fields:

- **leftThread**
- **rightThread**

If t→leftThread == true, then t→leftChild contains a thread, otherwise a pointer to left child. Similar for t→rightThread.

```
template <class T>
class ThreadedNode {
friend class ThreadedTree;
private:
    bool  leftThread;
    ThreadedNode * leftChild;
    T data;
    ThreadedNode * rightChild;
    bool  rightThread;
};
```

```
template <class T>
class ThreadedTree {
public:
    // Tree operations
…
private:
    ThreadedNode *root;
};
```

Let ThreadedInorderIterator be a nested class of ThreadedTree:

```
class ThreadedInorderIterator {
public:
    T* Next();
    ThreadedInorderIterator()
        { currentNode = root; }
private:
    ThreadedNode<T>* currentNode;
};
```

To make the left thread of the first node in inorder and the right thread of the last node in inorder un-dangle, we assume a head node for all threaded binary tree, let the two threads point to the head.

The original tree is the left subtree of the head, and the rightChild of head points to the head itself.

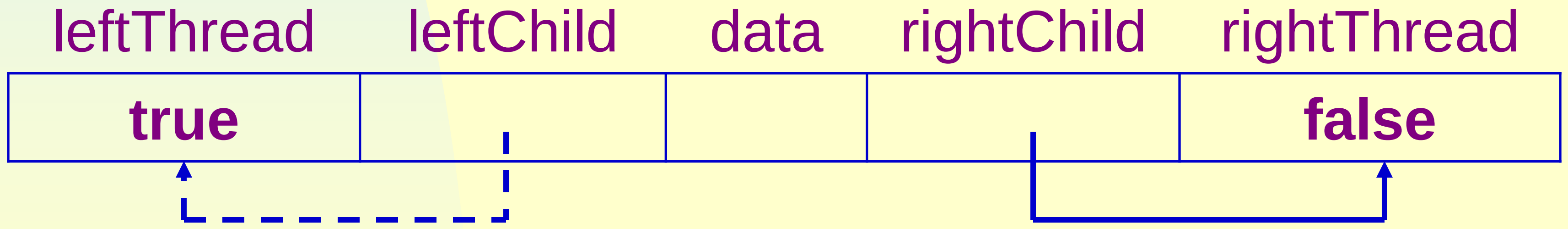


Fig.5.21 An empty threaded binary tree

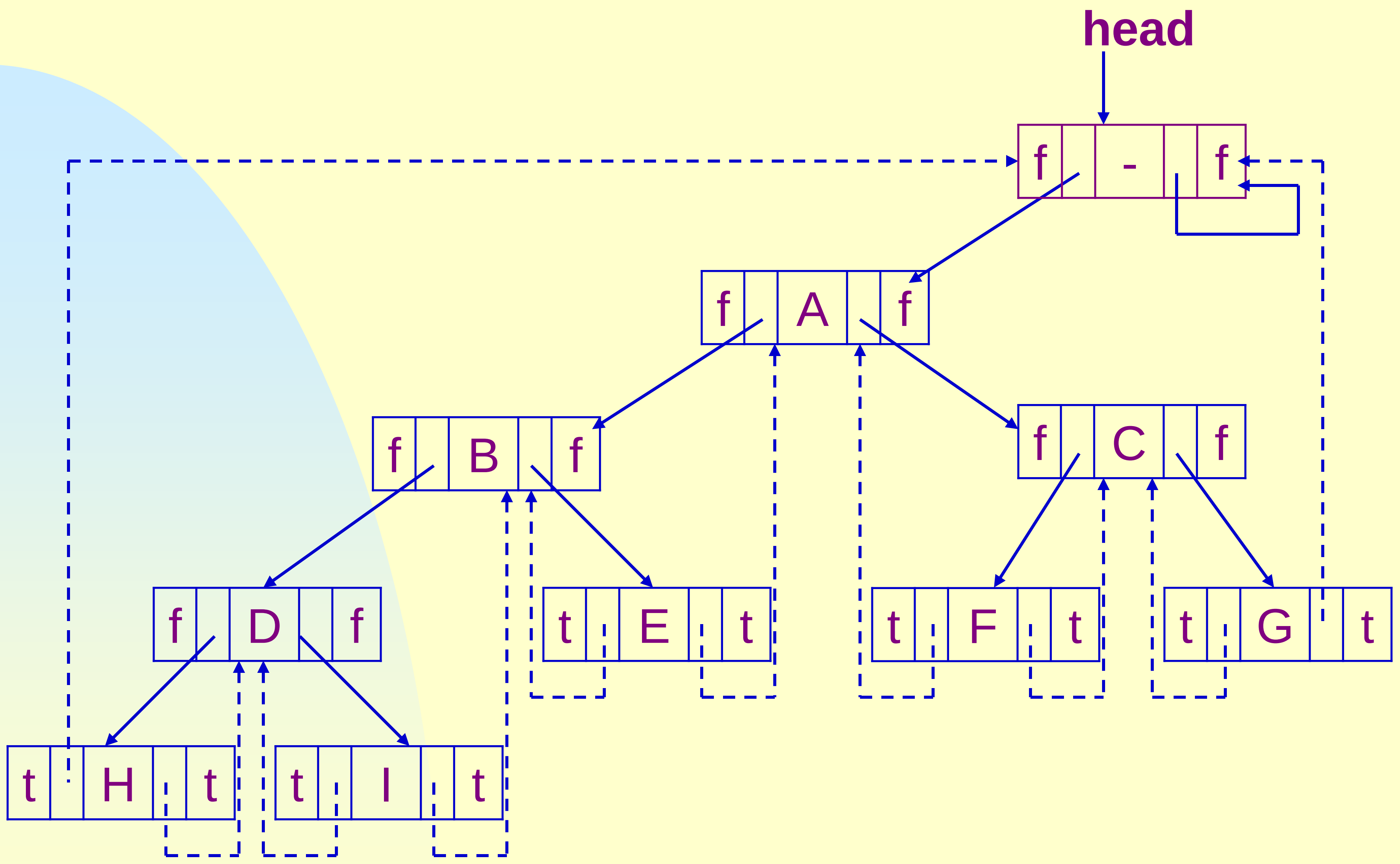


Fig.5.22 memory representation of threaded tree

we can see:

(1) The inorder predecessor of first node is the head node;

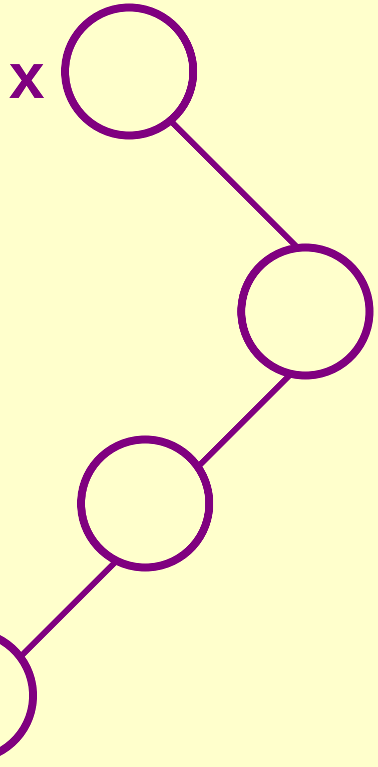
(2) The inorder successor of the last node is the head node.

5.5.2 Inorder Traversal of a Threaded Binary Tree

Observe:

(1) If x→rightThread==true, the inorder successor of x is x→rightChild;

(2) If x→rightThread==false, the inorder successor of x is obtained by following a path of leftChild from the right child of x until a node with leftThread==true is reached.



Thus we have:

```
T* ThreadedInorderIterator::Next()
{ // Return the inorder successor of currentNode in a threaded
  //  binary tree
     ThreadedNode<T>* temp=currentNode→rightChild;
     if (! currentNode→rightThread)
         while (!temp→leftThread) temp=temp→leftChild;
     currentNode=temp;
     if (currentNode==root) return 0; //no next
     else return &currentNode→data;
}
```

Note that when currentNode == root, Next() return the 1st node of inorder, thus we can use the following function to do an inorder travesal of a threaded binary tree:

```
template <class T>
void ThreadedTree::Inorder()
{
   ThreadedInorderiterator ti;
   for (T* p = ti.Next(); p; p = ti.Next())
   Visit(*p);
}
```

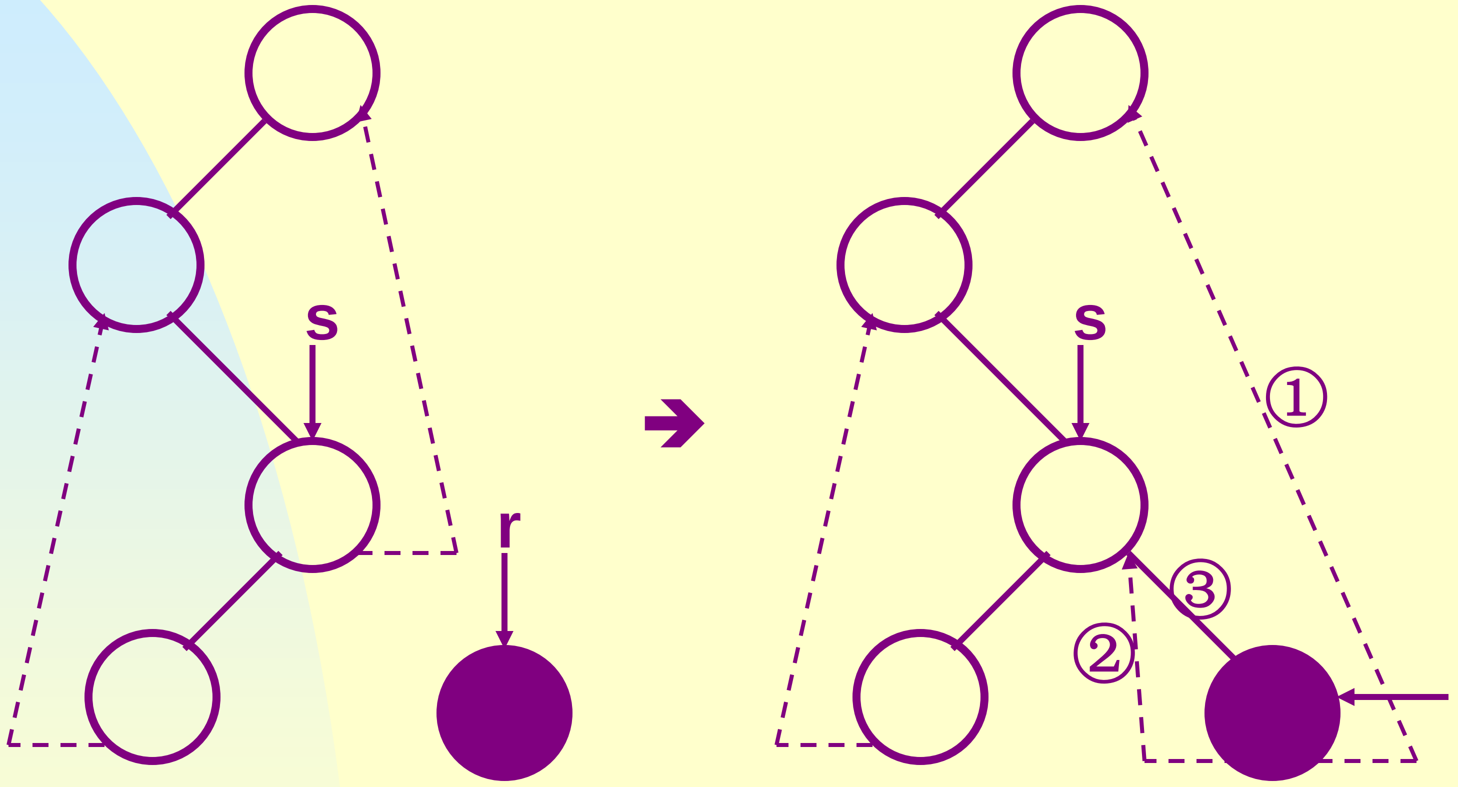
Since each node is visited at most 2 times, the computing time is readily seen to be O(n) for n nodes tree, and the traversal has been done without using a stack.

5.5.3 Inserting a Node into a Threaded Binary Tree

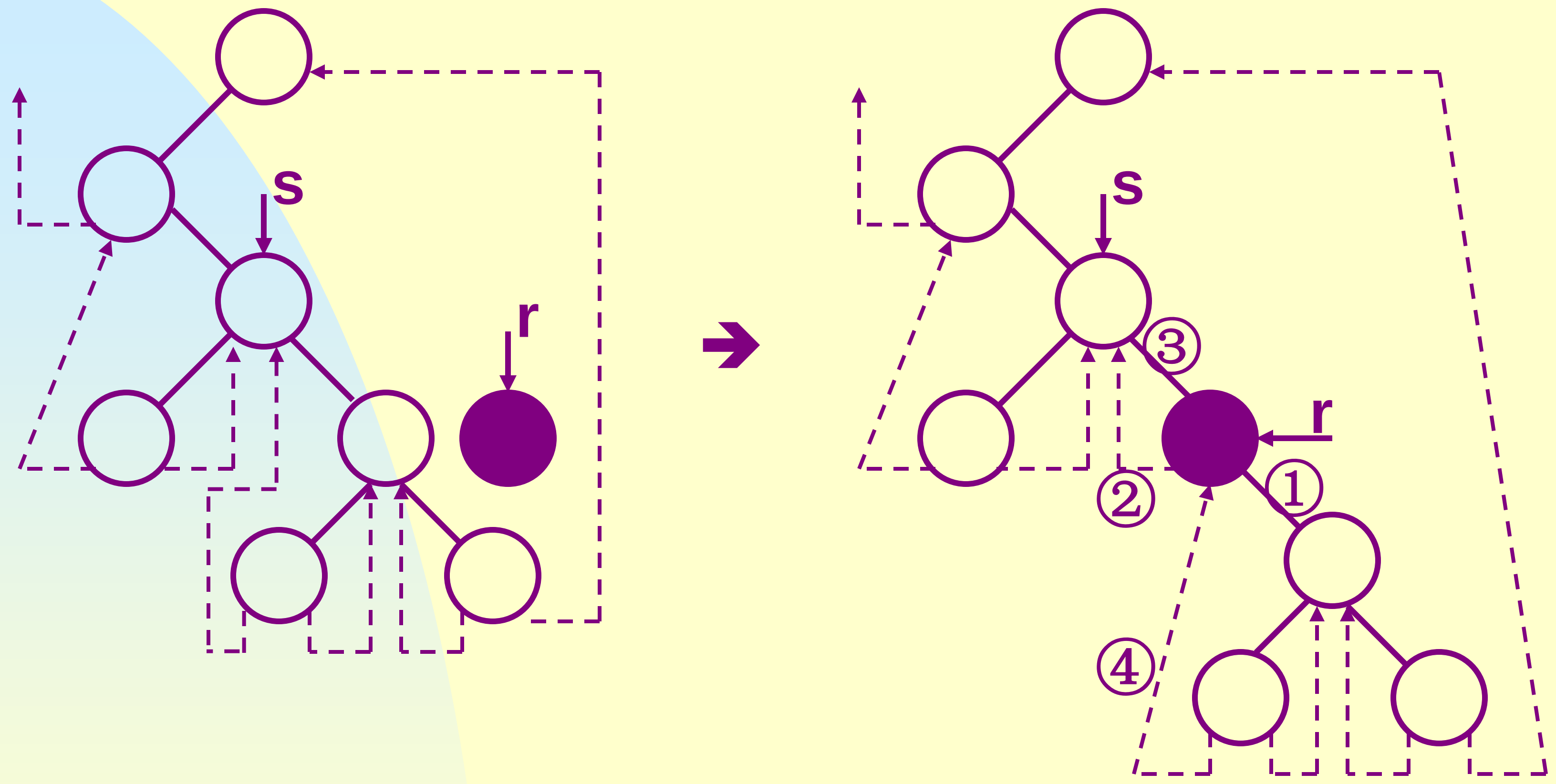
Insertion into a threaded tree provides the function for growing threaded tree.

We shall study only the case of inserting r as the right child of s. The left child case is similar.

(1) If s→rightThread==true, as:



(2) If s→rightThread==false, as:



In both (1) and (2), actions ①,②, ③ are the same, ④ is special for (2).

```
template <class T>
void  ThreadedTree<T>::InsertRight(ThreadedNode<T>* s,
                                   ThreadedNode<T>* r)
{ // insert r as the right child of s
    r→rightChild=s→rightChild;              // ①
    r→rightThread=s→rightThread;            // ① note s!=t.root,
    r→leftChild=s;                          // ②
    r→leftThread=true;                      // ②
    s→rightChild=r;                         // ③
    s→rightThread=false;                    // ③
    if (! r→rightThread) {     // case (2)
        ThreadedNode<T>* temp=InorderSucc(r);  // ④
        temp→leftChild=r;                      // ④
    }
}
```

Exercises: P277-1, P278-4

5.6 Heaps

5.6.1 Priority Queues

In a priority queue, the element to be deleted is the one with highest (or lowest) priority.

Assume type T is defined so that operators <, >, etc. compare element priorities.

ADT 5.2 MaxHeap

```
template <class T>
class MaxPQ {
public:
   virtual ~MaxPQ { }
      // virtual destructor
   virtual bool IsEmpty() const = 0;
      // return true iff the priority queue is empty
   virtual const T& Top() const = 0;
      // return reference to the max element
   virtual void Push(const T&) = 0;
      // add an element to the priority queue
   virtual void Pop() = 0;
      // delete the max element
};
```

The simplest way to represent a priority queue is as an unordered linear list:

- IsEmpty and Push in O(1)
- Top and Pop in O(n).

By using a max heap:

- IsEmpty and Top in O(1)
- Push and Pop in O(log n).

5.6.2 Definition of a Max Heap

Definition:

A max (min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any).

A max (min) heap is a complete binary tree that is also a max (min) tree.

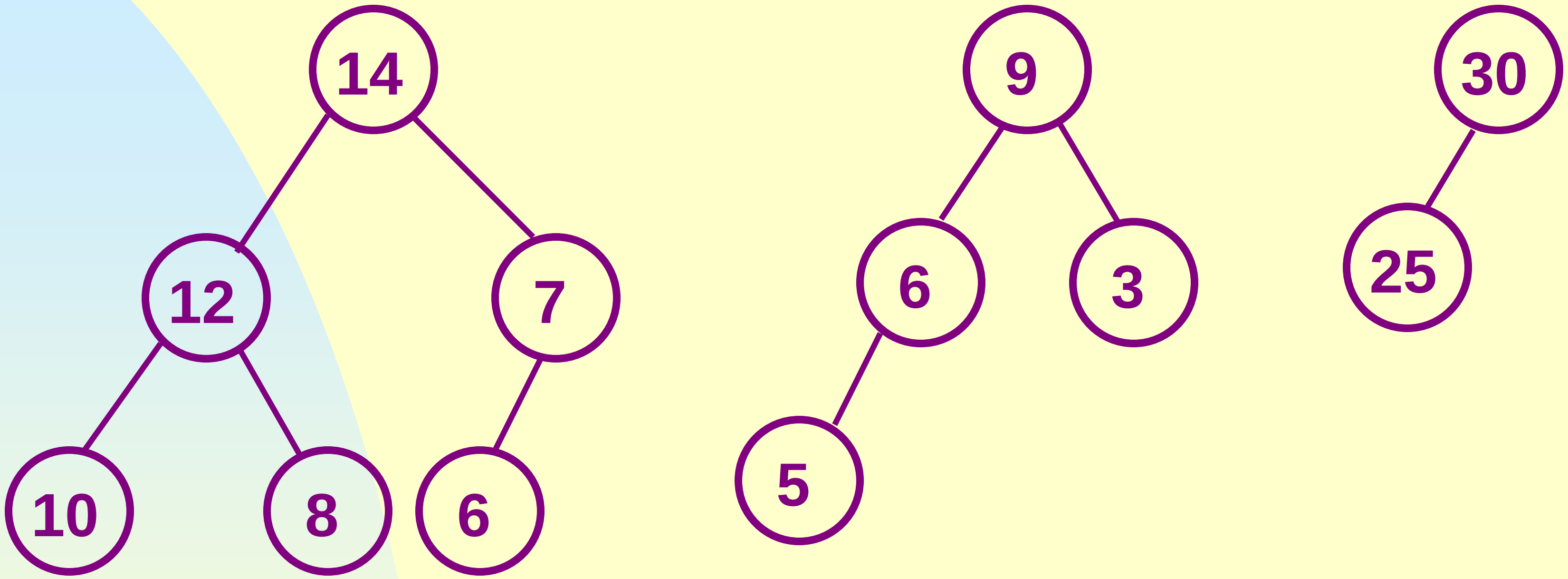


Fig. 5.24 Max heaps

Since max heap is a complete binary tree, we use array heap to represent it.

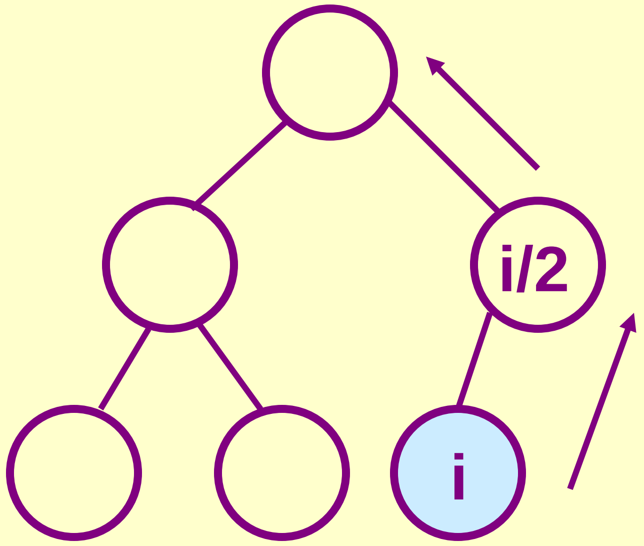
Thus we have the template MaxHeap class, which derives from MaxPQ<T>, as in the next slide:

```
template <class T>
class MaxHeap: public MaxPQ <T>
{
public:
     MaxHeap (int theCapacity=10);
     bool IsEmpty () { return heapSize==0;}
     const T& Top() const;
     void Push(const T&);
     void Pop();
private:
    T* heap;                // element array
    int heapSize;           // number of elements in heap
    int capacity;           // size of the array heap
};
```

```
template <class T>
MaxHeap<T>::MaxHeap (int theCapacity=10)
{ //constructor
    if (theCapacity < 1) throw "Capacity must be >= 1";
    capacity = theCapacity;
    heapSize = 0;
    heap = new T[capacity+1]; //heap[0] not used
}

template <class T>
Inline T& MaxHeap<T>::Top()
{
    if (IsEmpty()) throw "The heap is empty";
    return heap[1];
}
```

5.6.3 Insertion into a Max Heap

We use a **bubbling up** process to insert an element, begin at the node i=heapSize+1, and compare its key with that of its parent. If its key is larger, put its parent into node i , all the way up until it is no larger or reach the root.



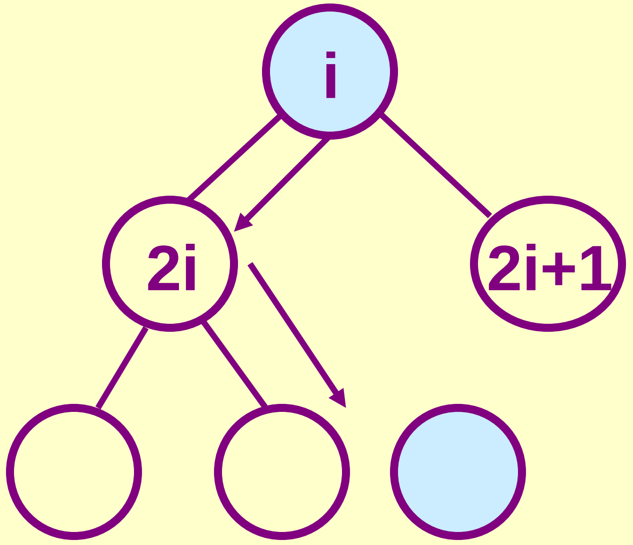
```
template <class T>
void MaxHeap<Type>::Push(const T& e)
{ // insert e into the max heap
    if (heapSize == capacity)  { // double the capacity
        ChangeSize1D(heap, capacity, 2*capacity);
        capacity *= 2;
    }
    int currentNode = ++heapSize;
    while (currentNode != 1 && heap[currentNode/2] < e)
    {  // bubble up
        heap[currentNode] = heap[currentNode/2];
        currentNode /=2;
    }
    heap[currentNode] = e;
}
```

Analysis of Push:

A complete binary tree with n nodes has a height $\lceil \log_2(n+1) \rceil$, the while loop is iterated O(log n) times, each takes O(1) time, so the complexity is O(log n).

5.6.4 Deletion from a Max heap

The max element is to be deleted from node 1, we assume the element in node heapSize to be in node 1, and heapSize--. Then we compare its key with that of its larger child. All the way down until it is no smaller or reach the leaf---trickle down.



```
template <class T>
void MaxHeap<T>::Pop()
{ // delete the max element.
   if (IsEmpty()) throw “Heap is empty. Cannot delete.”;
   heap[1].~T(); // delete the max

   // remove the last element from heap
   T lastE = heap[heapSize--];

   // trickle down
   int currentNode = 1;  // root
   int child = 2;      // left child of currentNode
```

```
    while (child <= heapSize)
    {
       // set child to the larger child of currentNode
       if (child<heapSize && heap[child]<heap[child+1]) child++;
       //child always points to the larger child

       // can we put lastE in currentNode?
       if (lastE>=heap[child]) break;  // yes

       // no
       heap[currentNode]=heap[child]; // move child up
       currentNode=child; child*=2; // move down a level
     }
    heap[currentNode]=lastE;
}
```

Analysis of Pop:

The height of a heap with n nodes is $\lceil \log_2(n+1) \rceil$, the while loop is iterated O(log n) times, each takes O(1) time, so the complexity is O(log n).

Exercises: P287-2, 3

5.7 Binary Search Trees

5.7.1 Definition

When arbitrary elements are to be searched or deleted, heap is not suitable.

A **dictionary** is a collection of pairs, each pair has a key and an associated element.

Although natural dictionaries may have several pairs with the same key, we assume here that **no two pairs have the same key.**

The specification of a dictionary is given as **ADT 5.3.**

ADT 5.3

```
template <class K, class E>
class Dictionary {
public:
    virtual bool IsEmpty () const = 0;
        // return true iff the dictionary is empty
    virtual pair<K,E>* Get(const K&) const = 0;
        // return pointer to the pair with specified key;
        // return 0 if no such pair
    virtual void Insert(const pair<K,E>&) = 0;
        // insert the given pair; if key is a duplicate
        // update associated element
    virtual void Delete(const K&) = 0;
        // delete pair with specified key
};
```

The pair can be defined as:

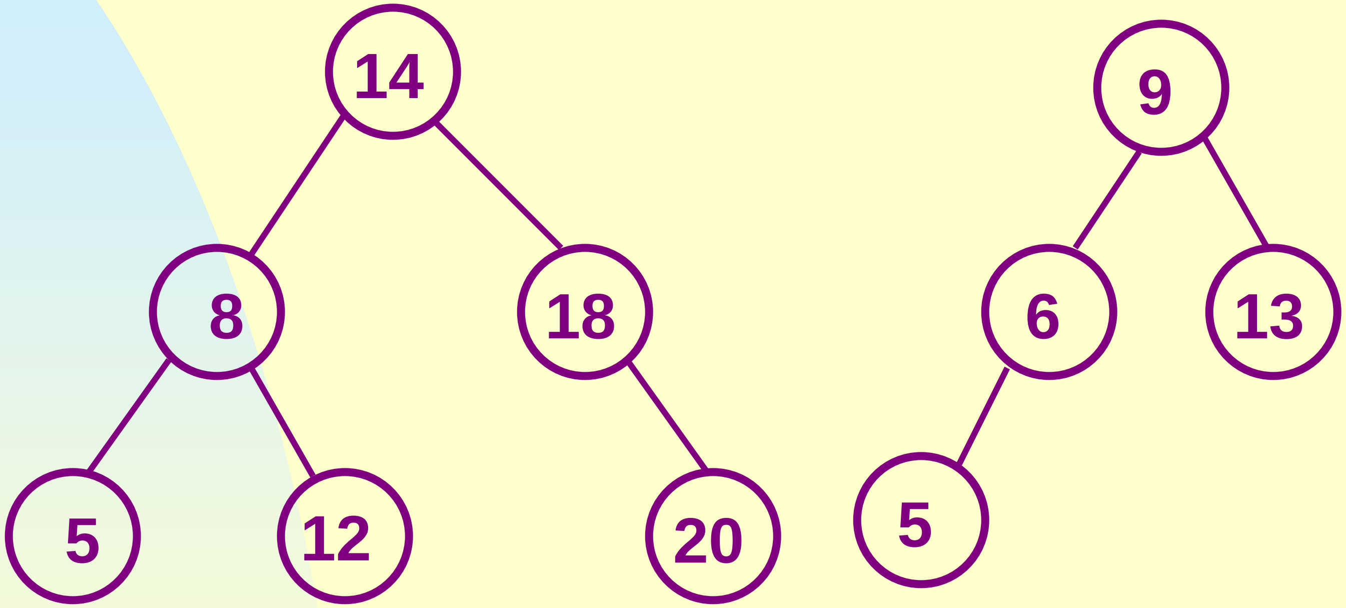
```
template <class K, class E>
struct pair
{
    K first;
    E second;
};
```

A binary search tree has a better performance when the functions to be performed are search, insert and delete.

Definition: A binary search tree, if not empty, satisfies the following properties:

(1) The root has a key.
(2) The keys (if any) in the left subtree are smaller than that in the root.
(3) The keys (if any) in the right subtree are larger than that in the root.
(4) The left and right subtrees are also binary search trees.

Note these properties imply that the keys must be distinct.



Binary search trees

5.7.2 Searching a Binary Search Tree

According to its properties, it is easy to search a binary search tree. Suppose search for an element with key k:

If k==the key in root, success;

If x<the key in root, search the left subtree;

If x>the key in root, search the right subtree.

Assume class BST **derives from the class** Tree<pair<K,E>>**.**

```
template <class K, class E> // Driver
pair<K,E>* BST<K,E>::Get(const K& k)
{ // Search *this for a pair with key k.
   return Get(root, k);
}

template <class K, class E> // Workhorse
pair<K,E>* BST<K,E>::Get(treeNode<pair<K,E>>* p,
                                                   const K& k)
{
    if (!p) return 0;
    if (k < p→data.first) return Get(p→leftChild, k);
    if (k > p→data.first) return Get(p→rightChild, k);
    return &p→data;
}
```

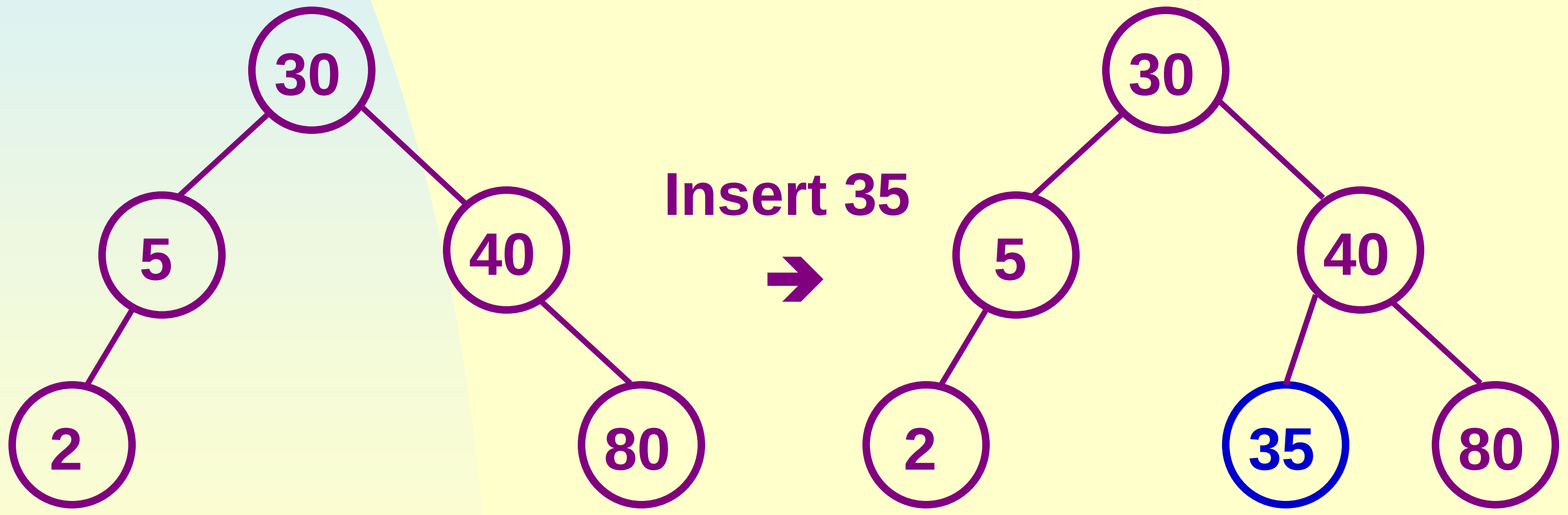
The recursive version can be easily changed into an iterative one as in the following:

```
template <class K, class E>  // Iterative version
pair<K,E>* BST<K,E>::Get(const K& k)
{
    TreeNode<pair<K,E>>* currentNode = root;
    while (currentNode)
       if (k < currentNode→data.first)
          currentNode = currentNode→leftChild;
       else if (k > currentNode→data.first)
          currentNode = currentNode→rightChild;
       else return &currentNode→data;

    // no matching pairs
    return 0;
}
```

As can be seen, a binary search tree of height h can be search by key in O(h) time.

5.7.3 Insertion into a Binary Search Tree

To insert a new element, search is carried out, if unsuccessful, then the element is inserted at the point the search terminated.



When the dictionary already contains a pair with key k, we simply update the element associated with this key to e.

```
template <class K, class E>
void BST<K,E>::Insert(const pair<K,E>& thePair)
{ // Insert thePair into the binary search tree
      // search for thePair.first, pp is parent of p
      TreeNode<pair<K,E>> *p=root, *pp=0;
      while (p) {
        pp=p;
        if (thePair.first < p→data.first) p=p→leftChild;
        else if (thePair.first > p→data.first) p=p→rightChild;
        else // duplicate, update associated element
             {p→data.second=thePair.second;return;}
      }
```

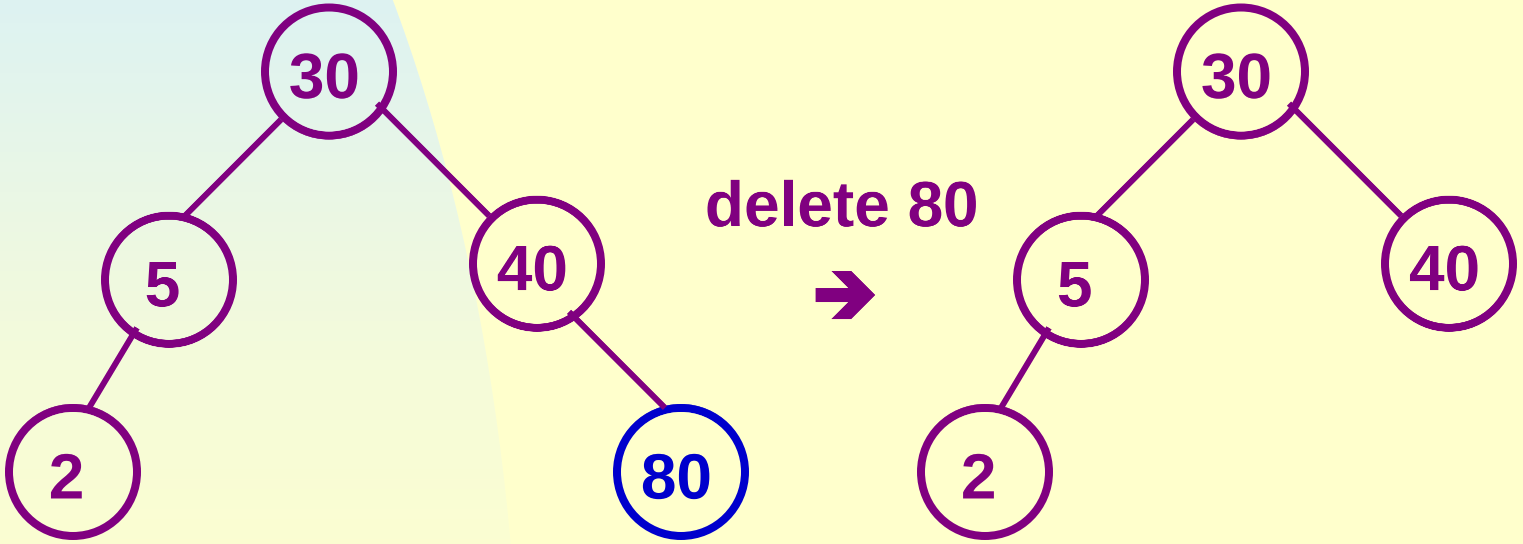
```
    // perform insertion
    p=new TreeNode<pair<K,E>>(thePair,0,0);
    if (root) // tree not empty
      if (thePair.first < pp→data.first) pp→leftChild=p;
        else pp→rightChild=p;
    else root=p;
}
```

The time is O(h).

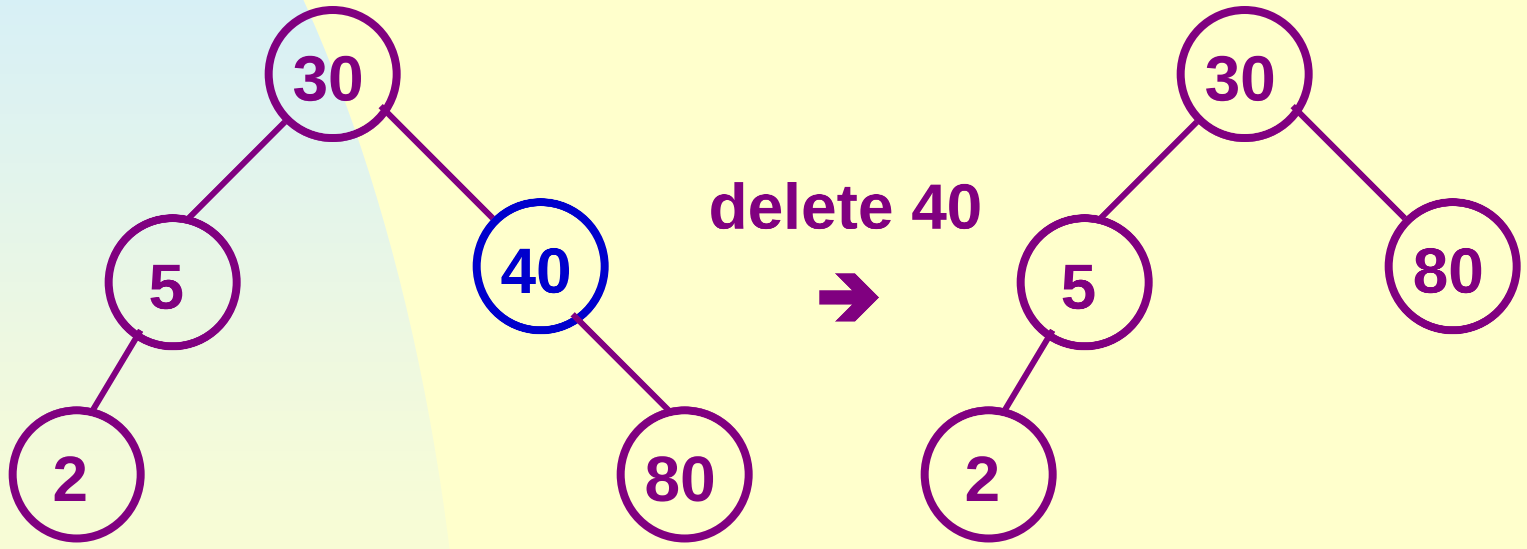
5.7.4 Deletion from a Binary Search Tree

3 cases:

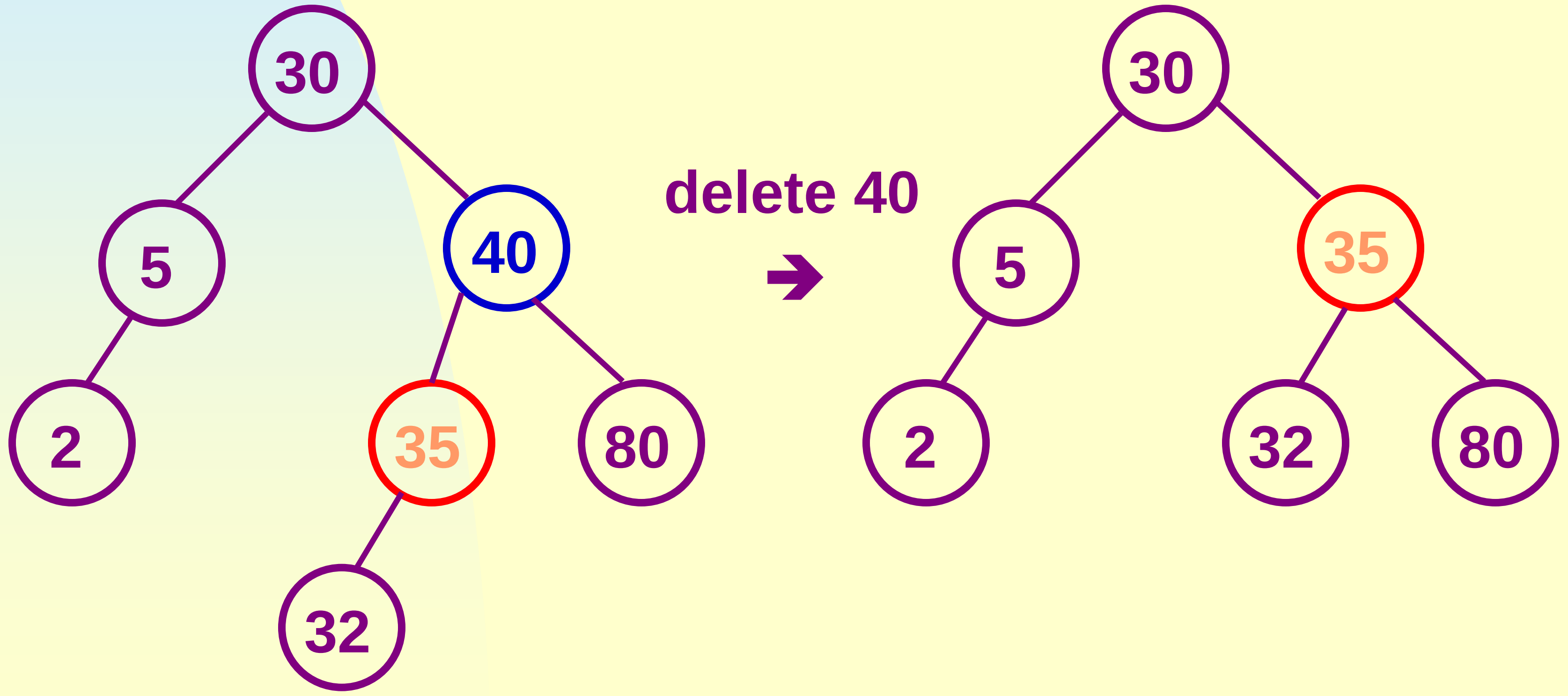
(1) deletion of a leaf: easy, set the corresponding child field of its parent to 0 and dispose the node.



(2) deletion of a nonleaf node with one child: easy, the child of the node takes the place of it and the node disposed.



(3) deletion of a nonleaf node with two children: the element is replaced by either the largest elements in its left subtree or the smallest element in its right subtree. Then delete that node, which has at most one child, in the subtree.



Exercises: P296-1,2

5.8 Selection Trees

5.8.1 Introduction

To merge k ordered sequences, called **runs**, we need to find the smallest from k possibilities, output it, and replace it with the next record in the corresponding run, then find the next smallest, and so on.

The most direct way is to make k-1 comparisons. For k>2, if we make use of the knowledge obtained from the last comparisons, the number of comparisons to find the next can be reduced.

Selection trees including winner tree and loser tree are suitable for this.

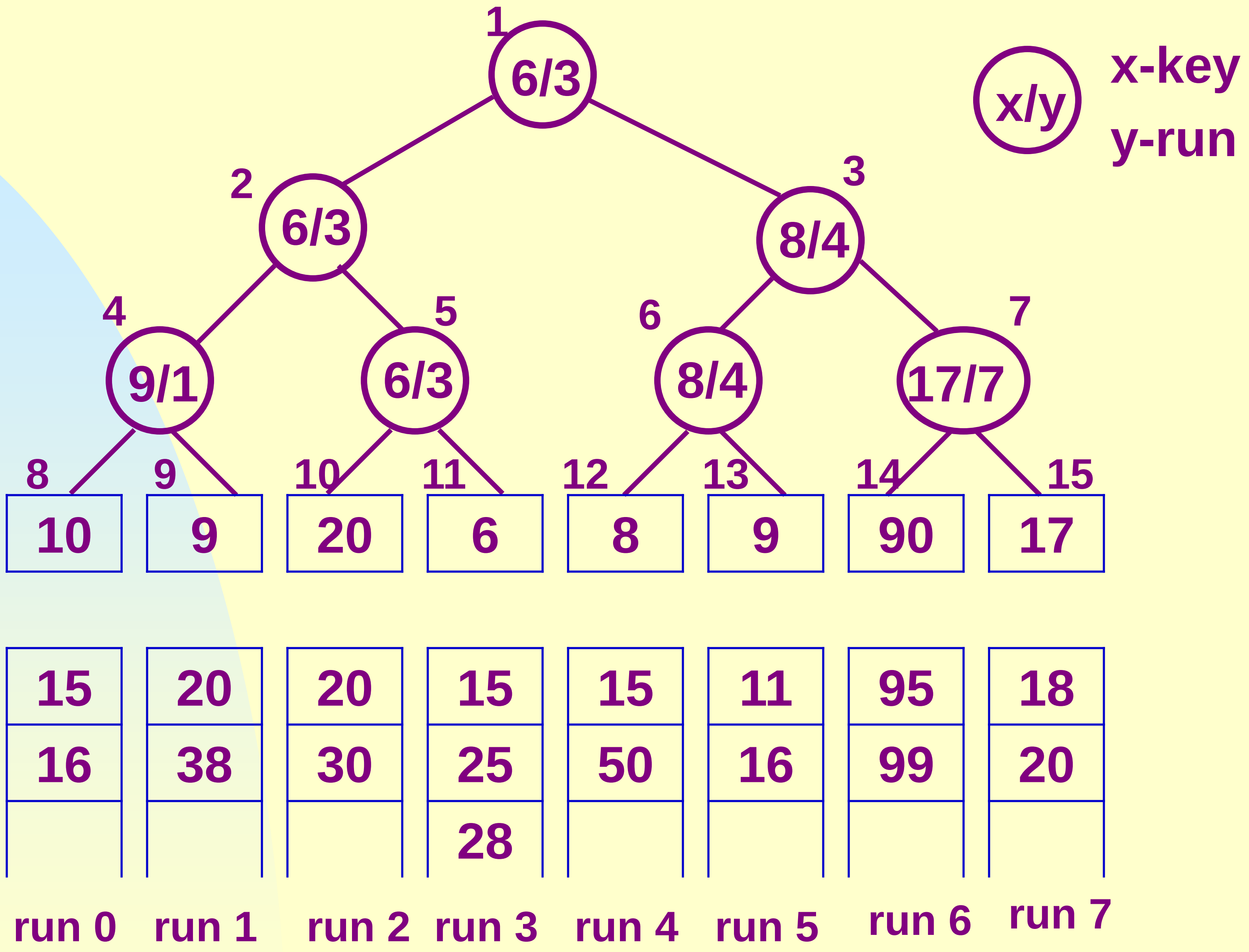
5.8.2 Winner trees

A winner tree is a complete binary tree in which each node represents the smaller (winner) of its children. The root represents the smallest.

The construction of a winner tree may be compared to the playing of a tournament.

Leaf node---the first record in the corresponding run.

Nonleaf node---contains an index to the winner of a tournament.

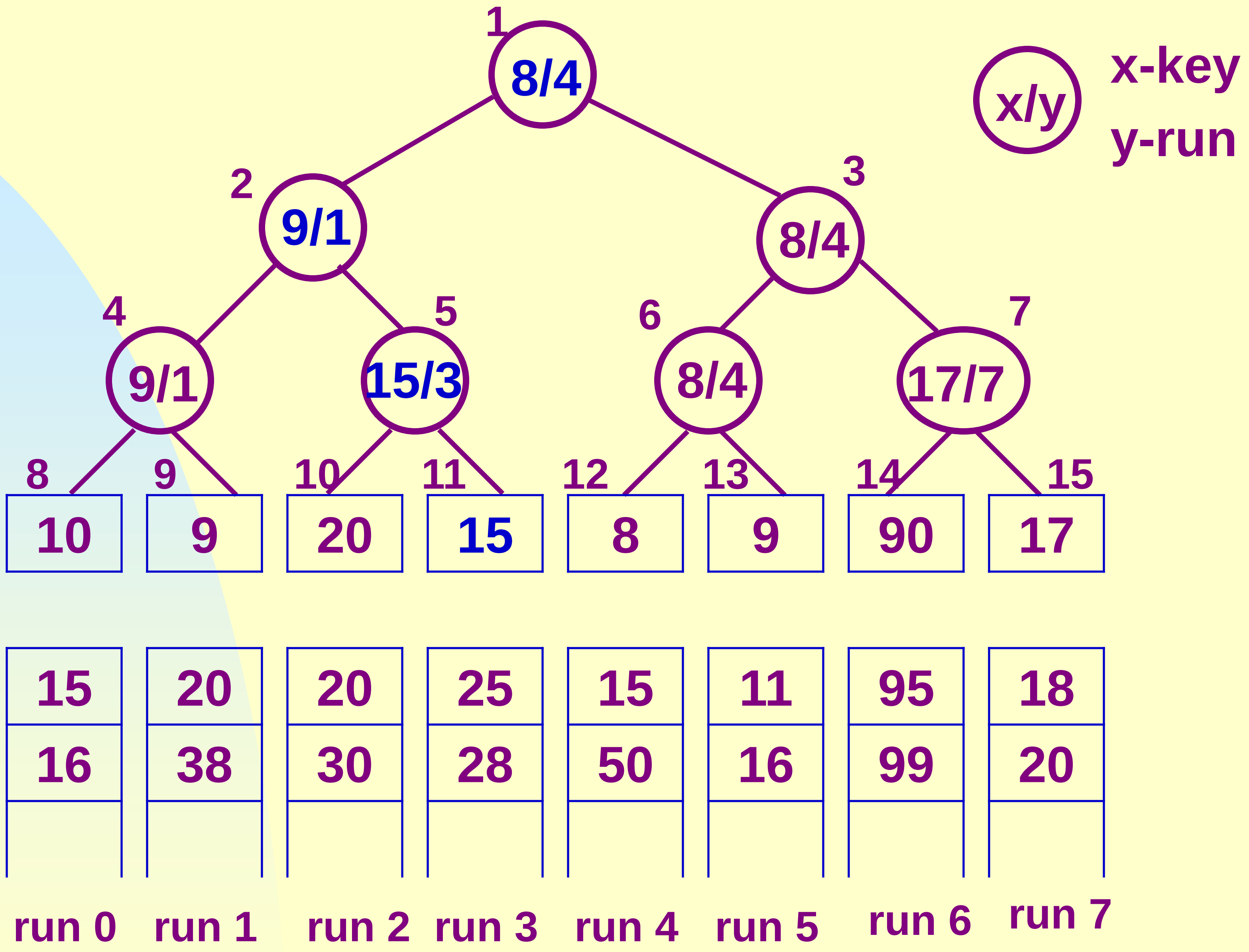

1
6/3
2
6/3
3
8/4
4
9/1
5
6/3
6
8/4
7
17/7
8
9
10
11
12
13
14
15
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
28
15
50
11
16
95
99
18
20
run 0
run 1
run 2
run 3
run 4
run 5
run 6
run 7
x/y
x-key
y-run

Actually, leaf nodes can be record buffer[0] to buffer[k-1]. The node number of Buffer[i] in the tree is k+i.

Now record pointed by the root (key==6) is output, buffer[3] is empty, the next record from run 3 (key ==15) is input to buffer[3].

To reconstruct the tree, tournament has to be played along the path from node 11 to the root.

As in the next slide:


1
8/4
2
9/1
3
8/4
4
9/1
5
15/3
6
8/4
7
17/7
8
9
10
11
12
13
14
15
10
9
20
15
8
9
90
17
15
20
20
25
15
11
95
18
16
38
30
28
50
16
99
20
run 0
run 1
run 2
run 3
run 4
run 5
run 6
run 7
x/y
x-key
y-run

The tournament is played between sibling nodes and the result put into the parent node. Lemma 5.4 may be used to compute the address of sibling and parent nodes efficiently.

Time of reconstruction: $O(\log_2 k)$

Time of setting up the tree: $O(k)$

Thinking how to set up the initial winner tree.

Setting up a initial winner tree with 2k-1 nodes, k of which are leaves. Thus there are k-1 times of comparison to complete the task.

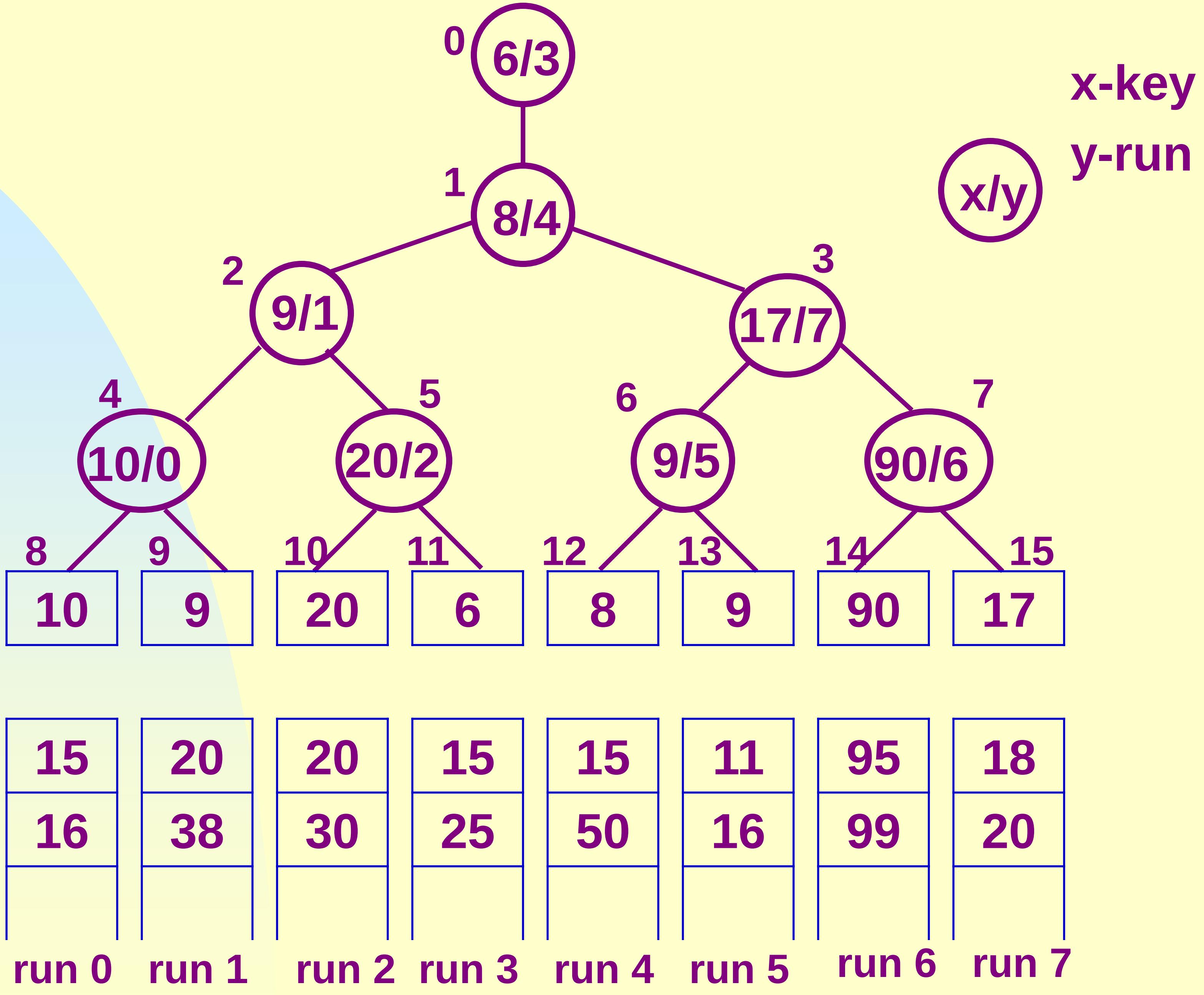
5.8.3 Loser Trees

To simplify the restructuring we can use a loser tree.

A selection tree in which **each nonleaf node retains a pointer to the loser** is called a loser tree.

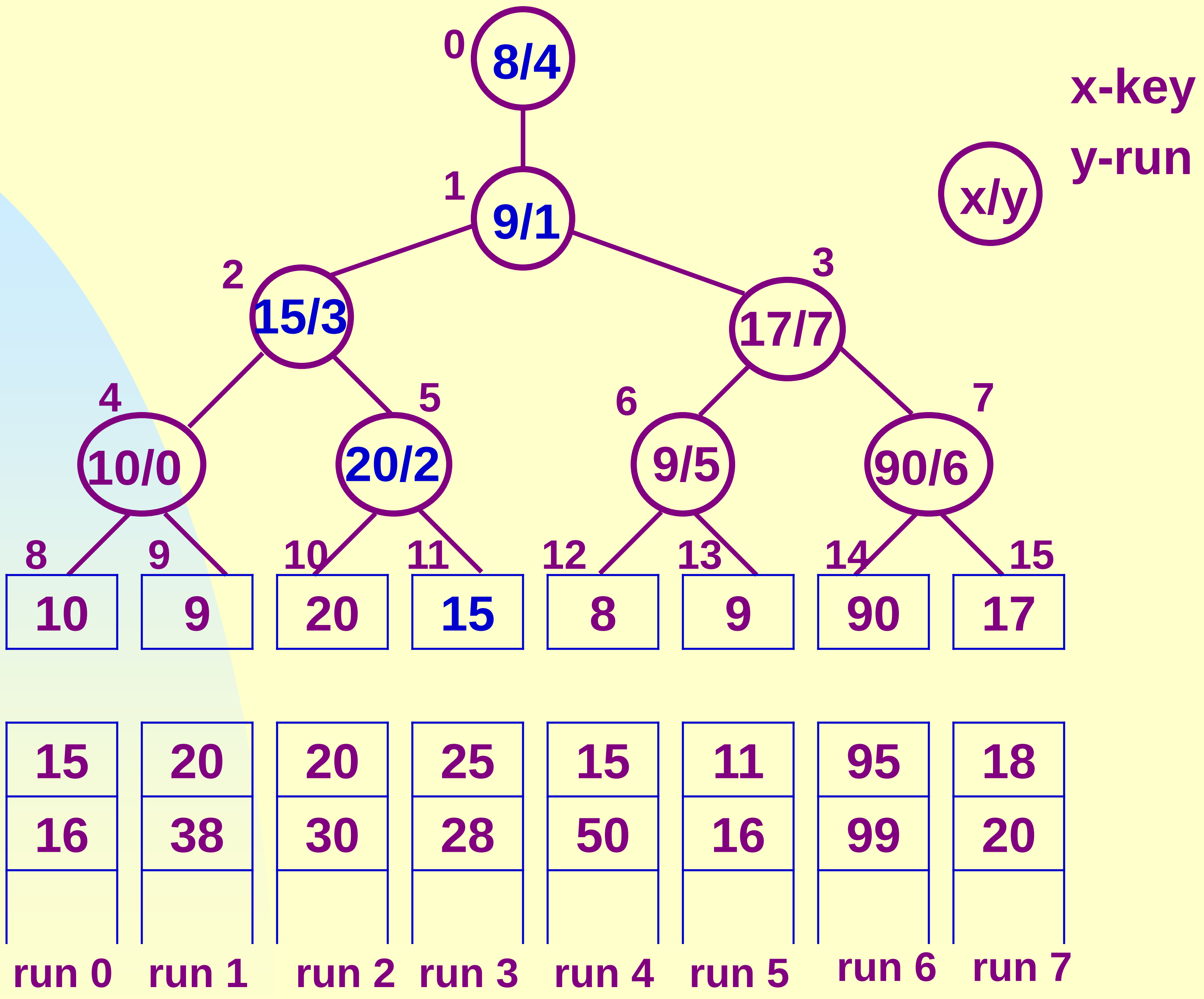
Actually, the loser in node p is the loser of the winners of the children of p.

The next slide shows a loser tree corresponding the previous winner tree.


0
6/3
1
8/4
2
9/1
3
17/7
4
10/0
5
20/2
6
9/5
7
90/6
8
9
10
11
12
13
14
15
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
95
99
18
20
run 0
run 1
run 2
run 3
run 4
run 5
run 6
run 7
x/y
x-key
y-run

Node 0 represents the overall winner.

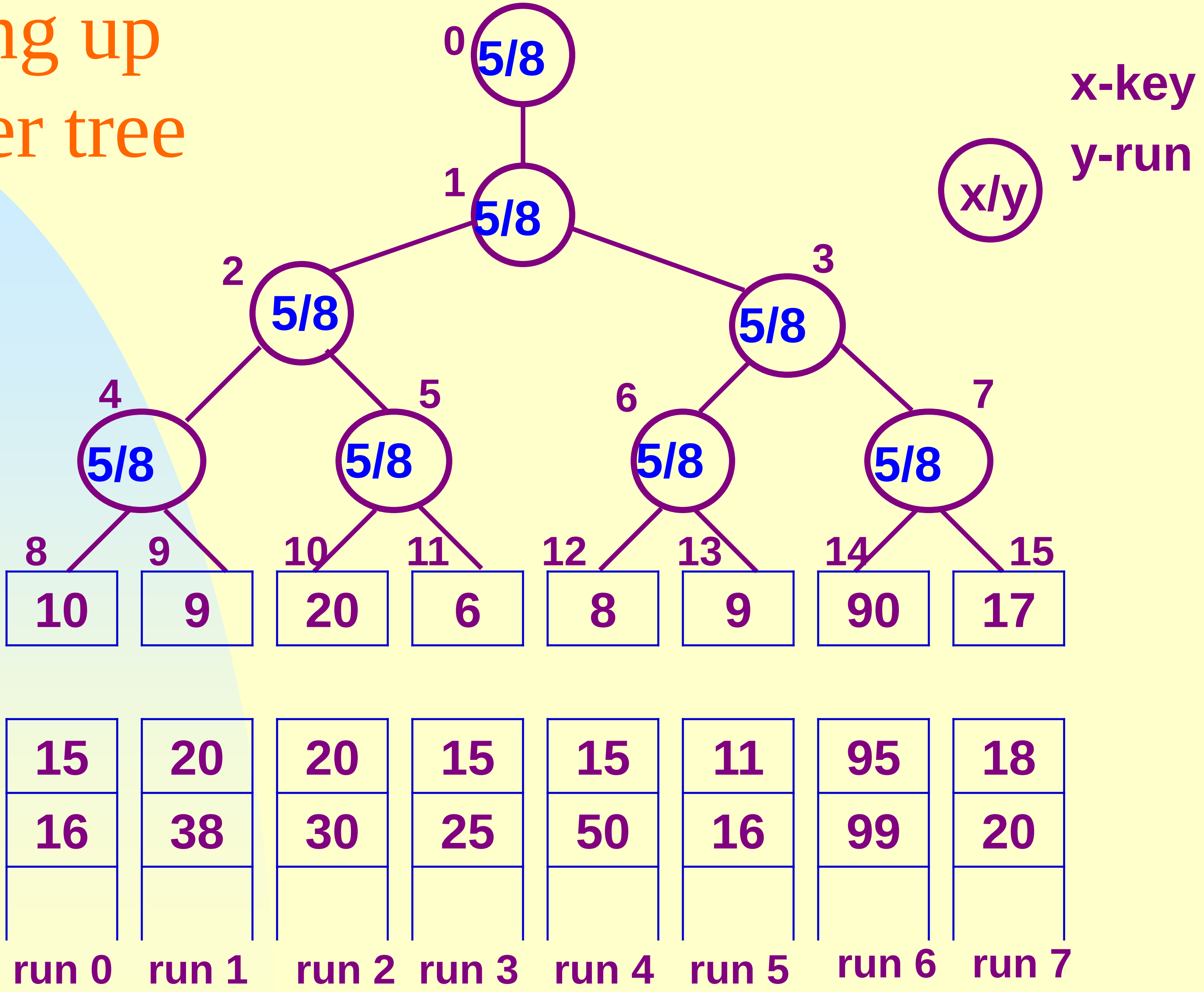
After outputting record[3] (node 11, key 6), the tree is restructured by reading next record from run 3 and playing tournament along the path from node 11 to node 1, as in the next slide:


0
8/4
1
9/1
2
15/3
3
17/7
4
10/0
5
20/2
6
9/5
7
90/6
8
9
10
11
12
13
14
15
10
9
20
15
8
9
90
17
15
16
20
38
20
30
25
28
15
50
11
16
95
99
18
20
run 0
run 1
run 2
run 3
run 4
run 5
run 6
run 7
x/y
x-key
y-run

The records with which these tournaments are to be played are readily available from the parent nodes. As a result, sibling nodes along the path from 11 to 1 are not accessed.

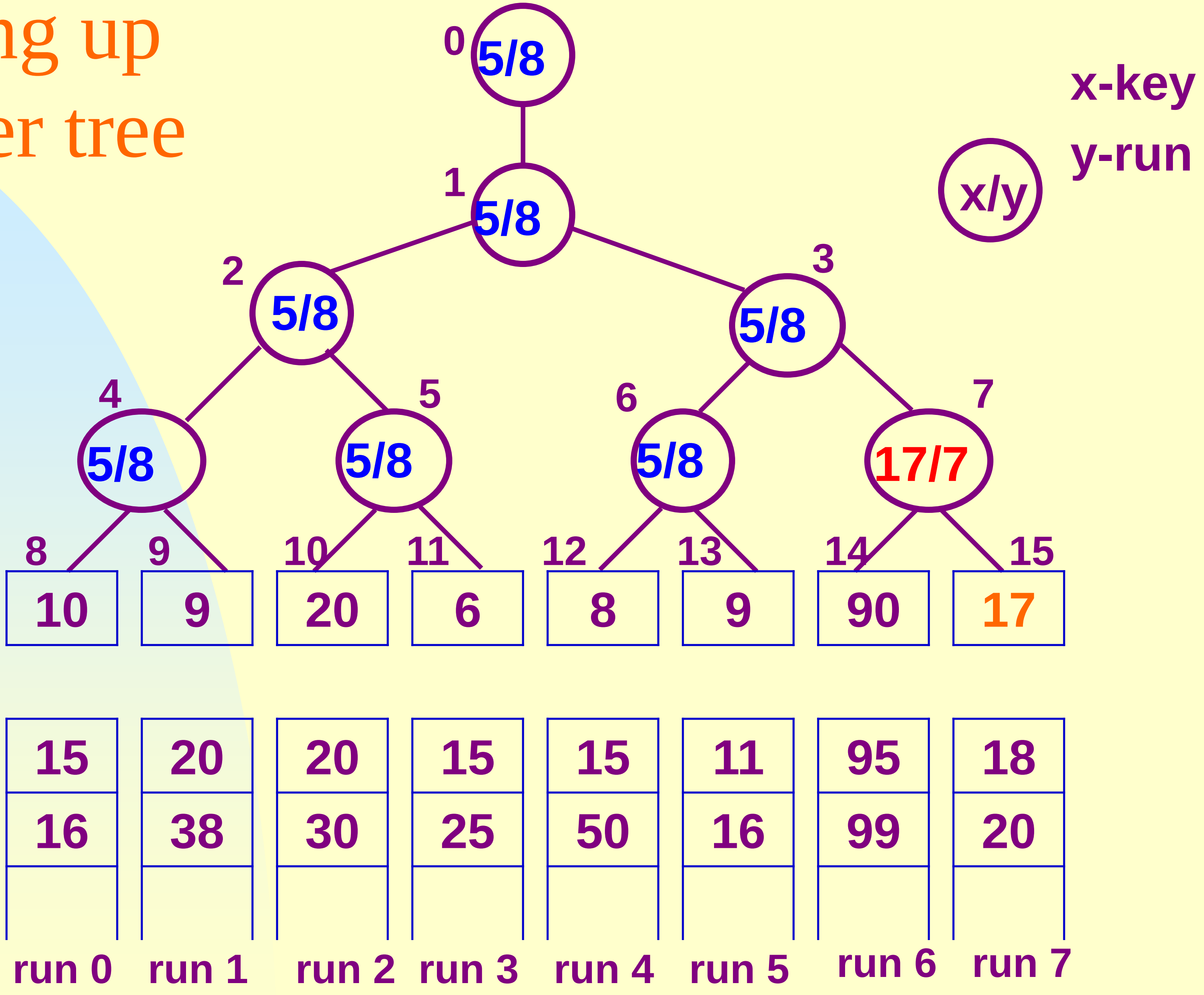
Exercises: P301-1,4

Setting up a loser tree

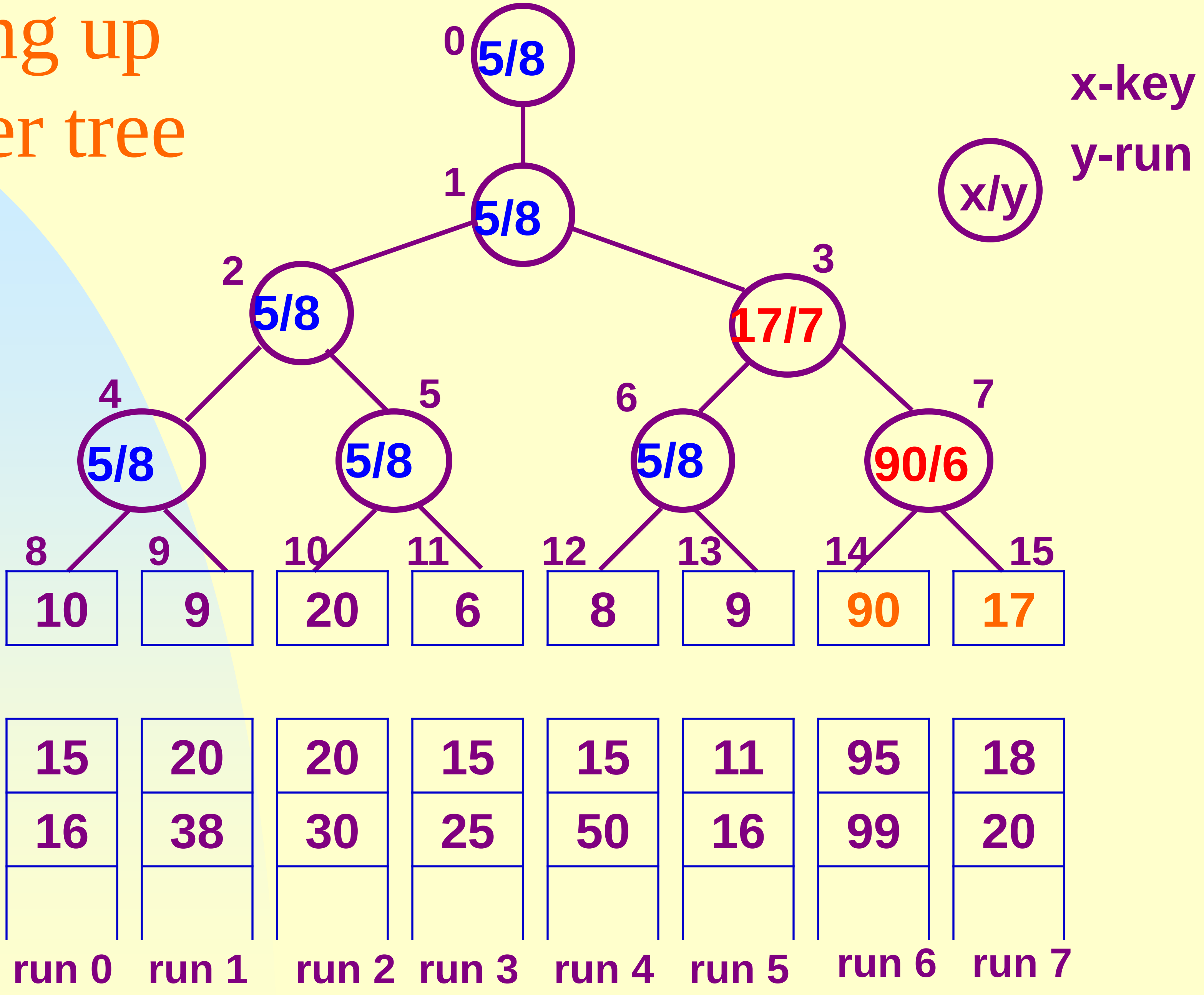


5 is less than the minimal key

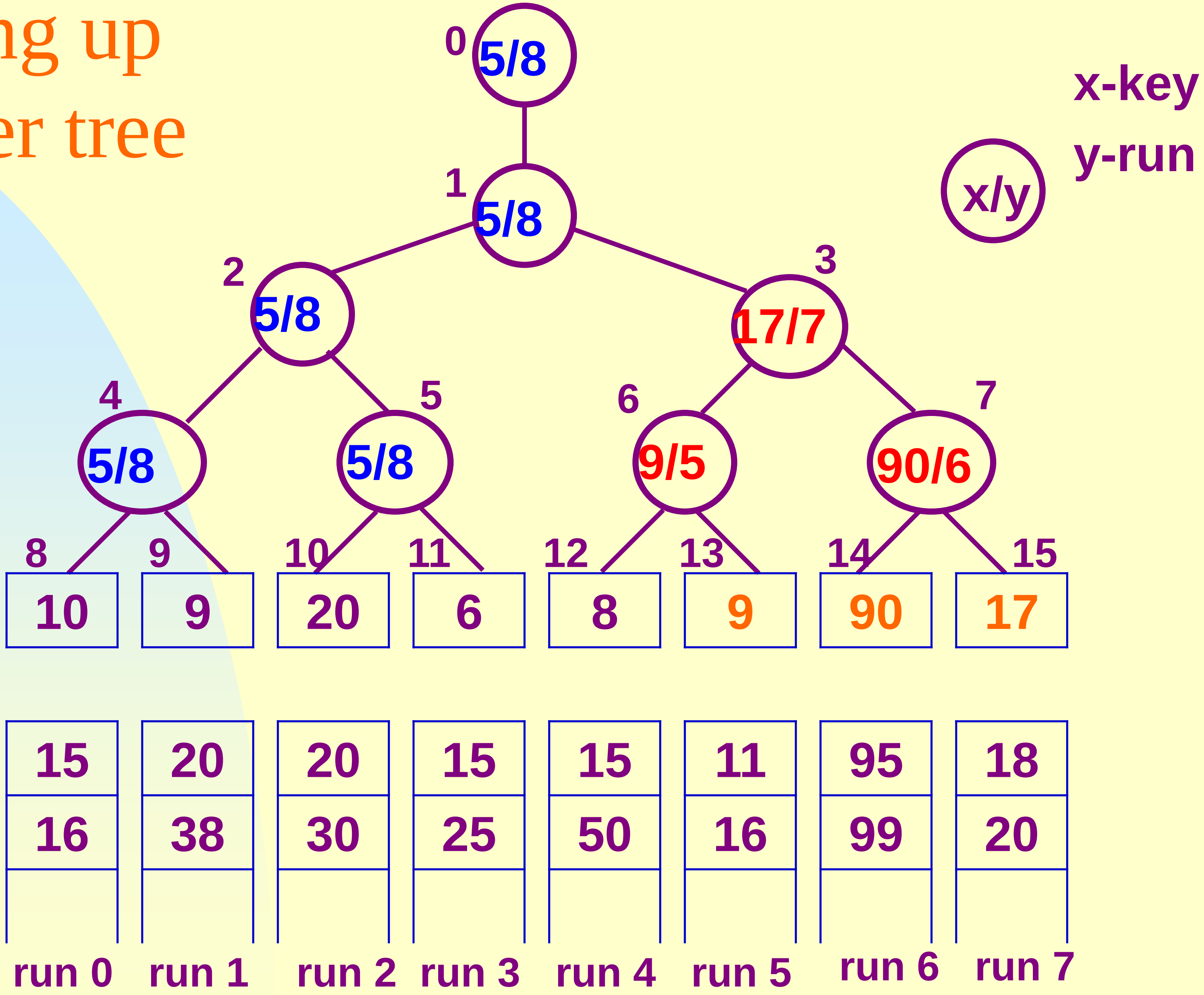
Setting up a loser tree



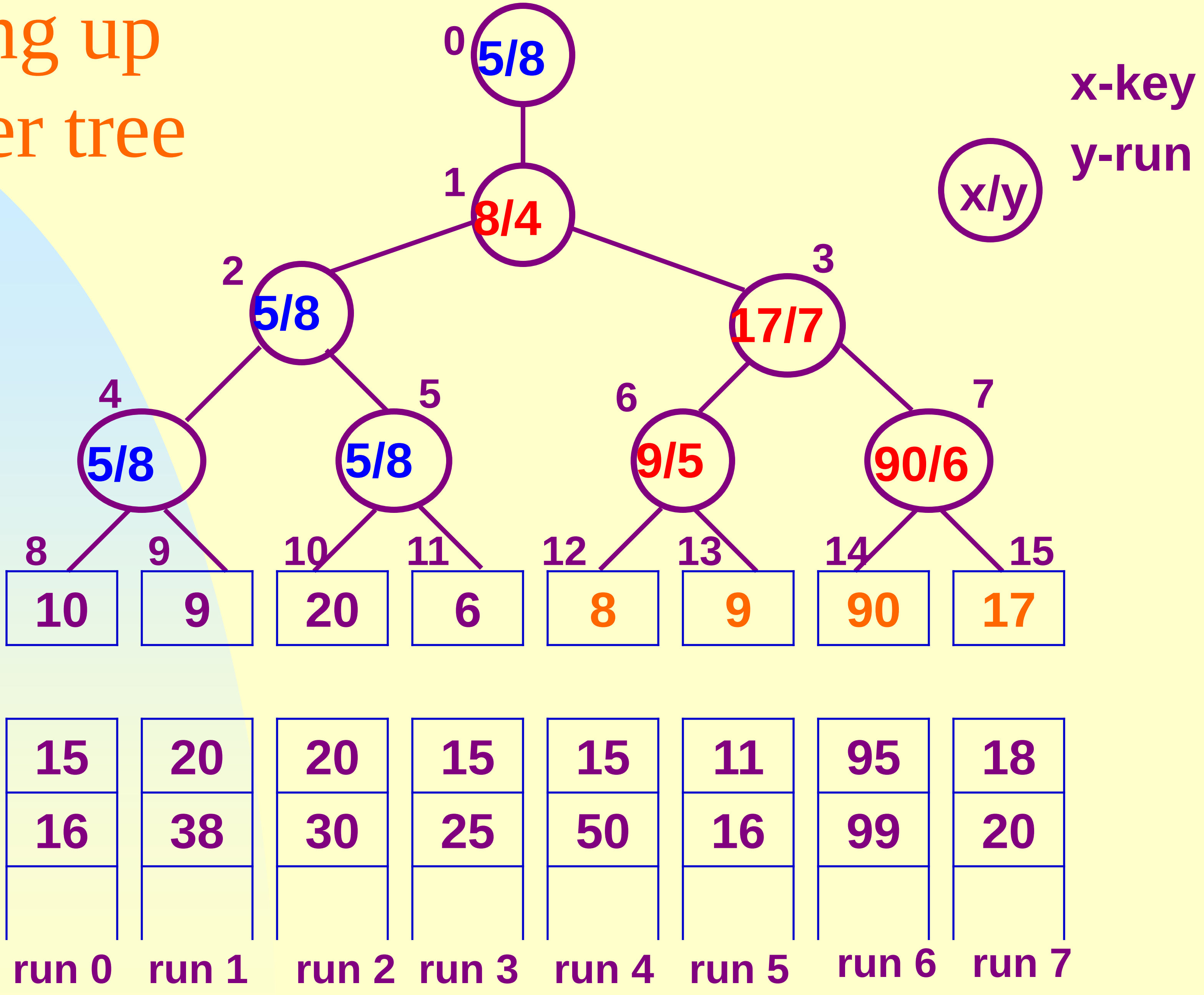
Setting up a loser tree



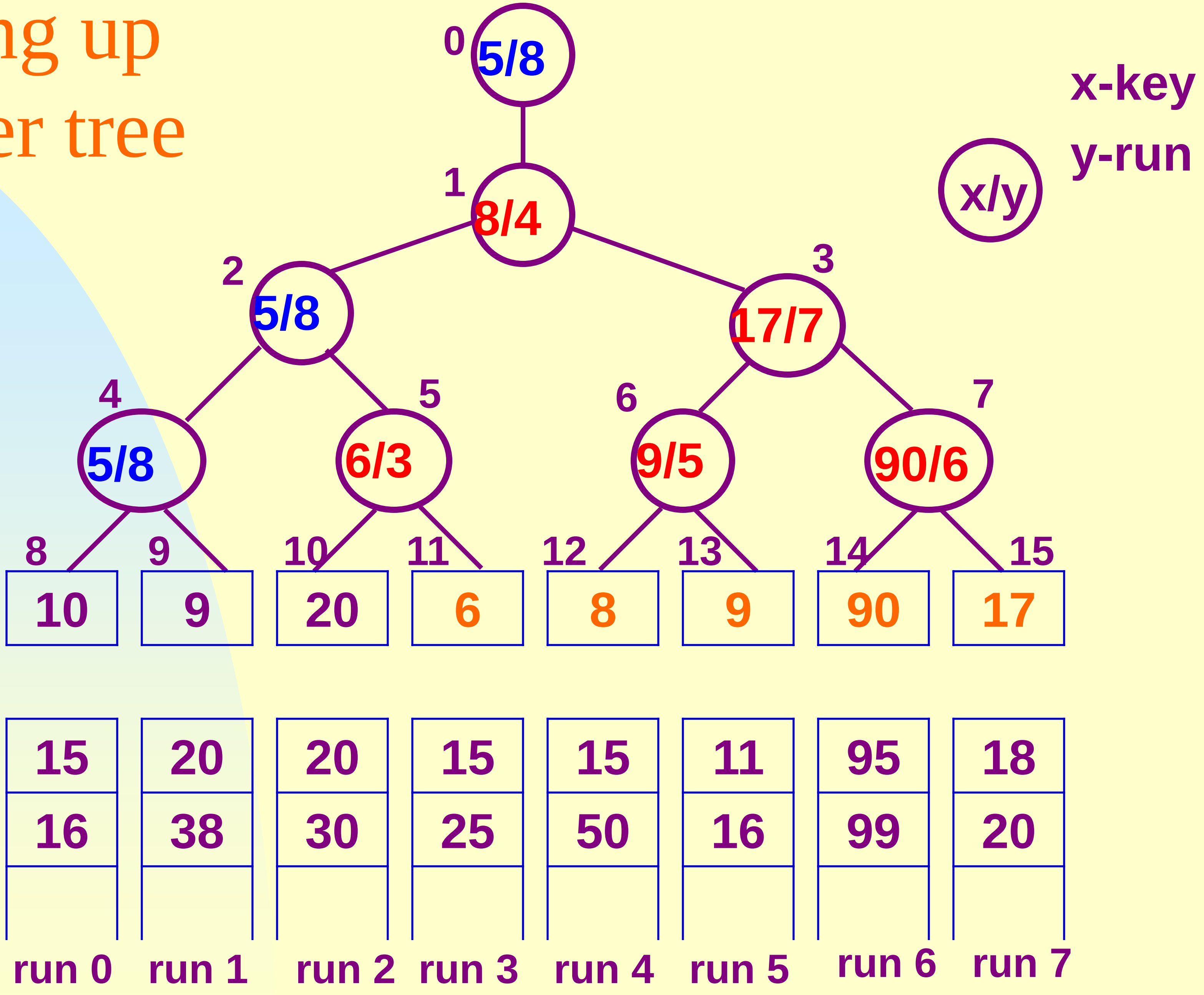
Setting up a loser tree



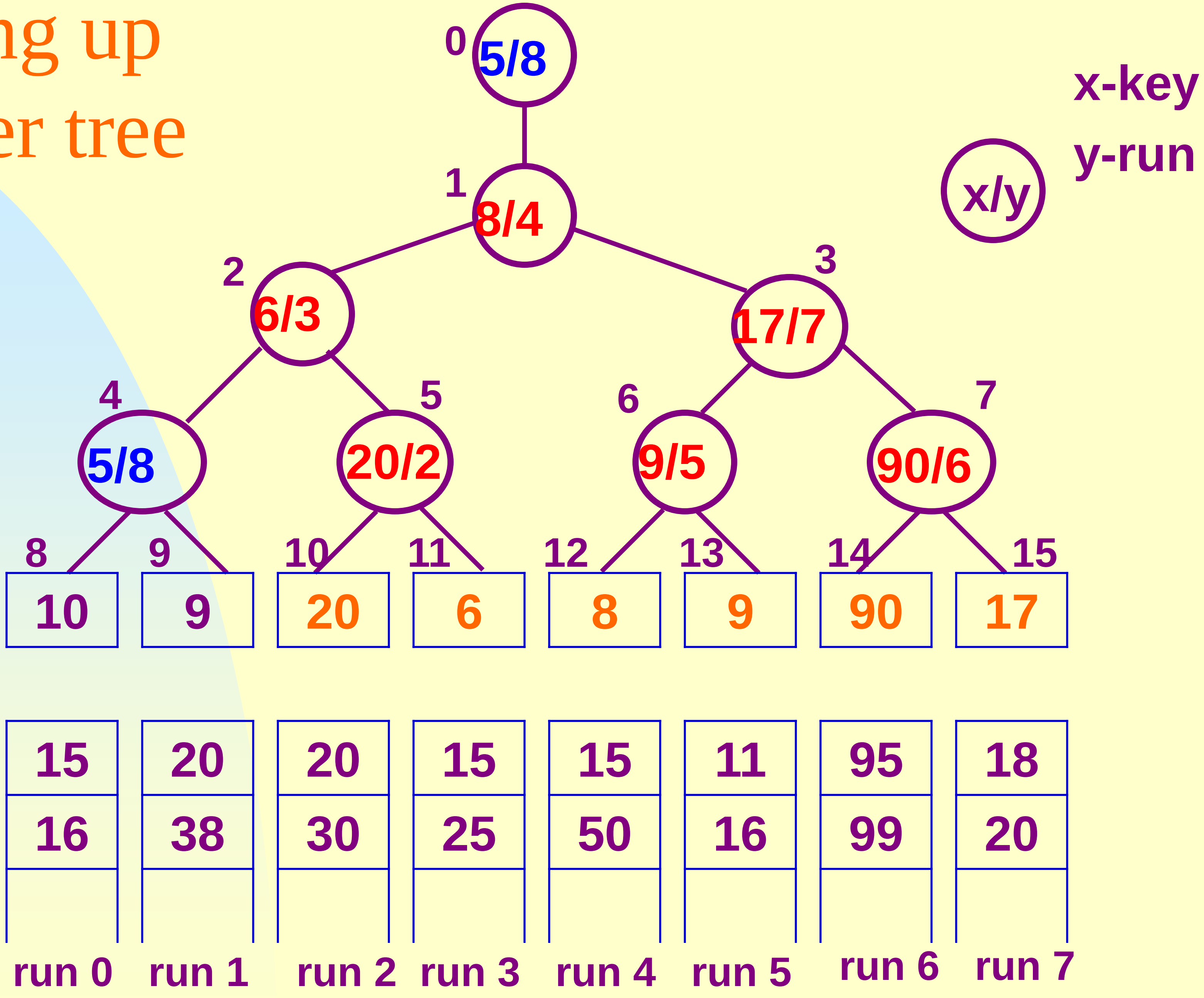
Setting up a loser tree



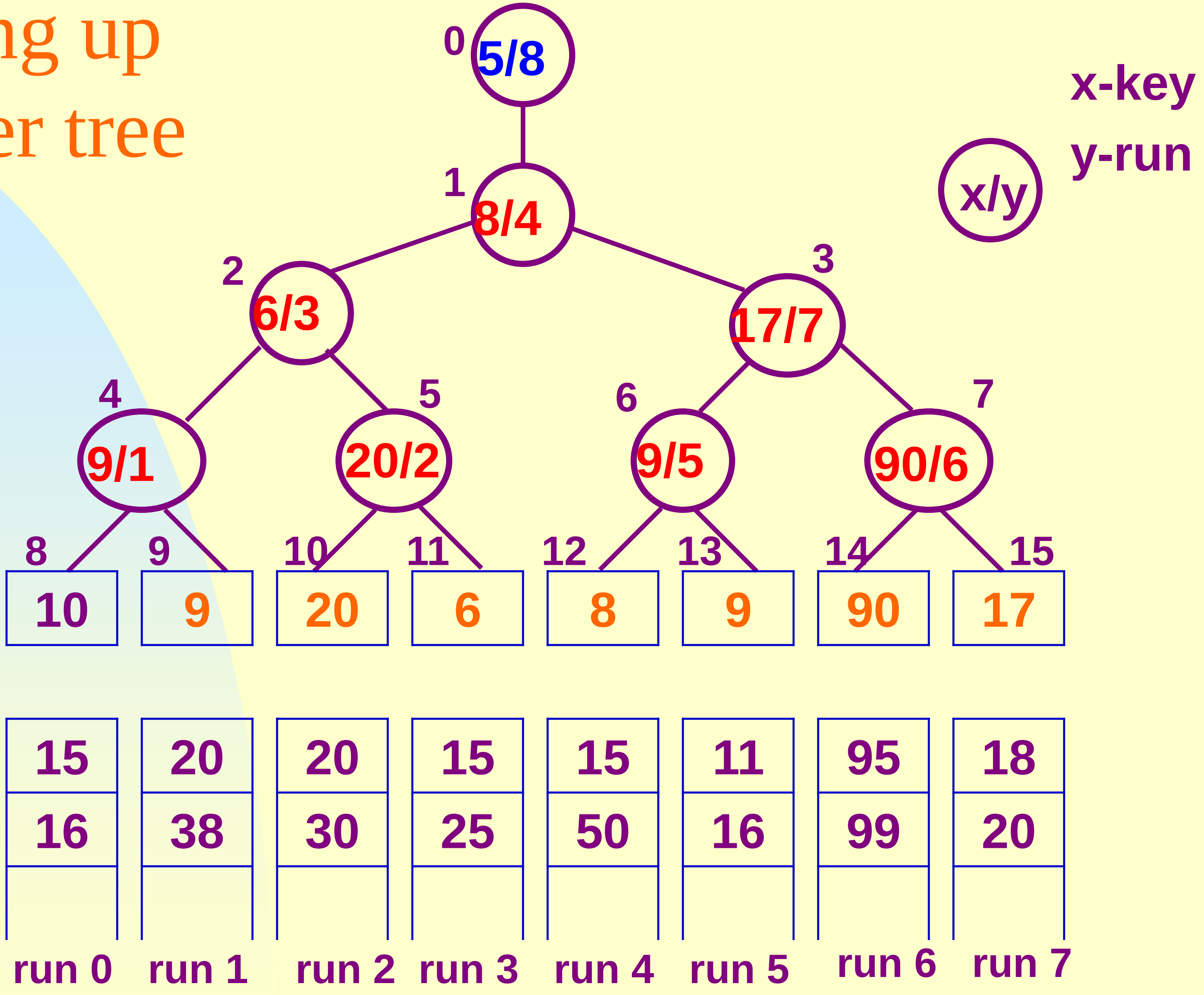
Setting up a loser tree



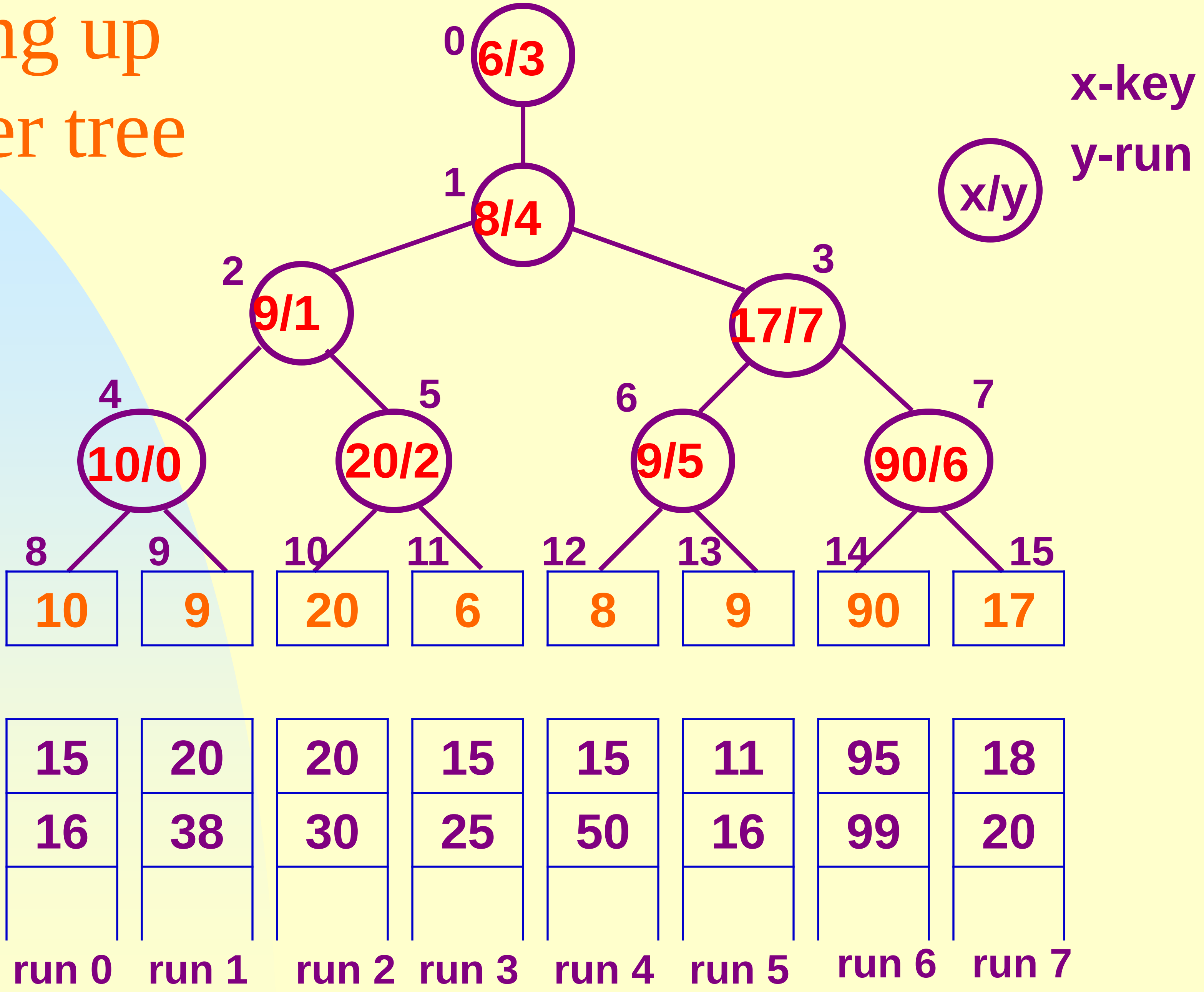
Setting up a loser tree



Setting up a loser tree



Setting up a loser tree



5.9 Forests

Definition: A forest is a set of n≥0 disjoint trees.

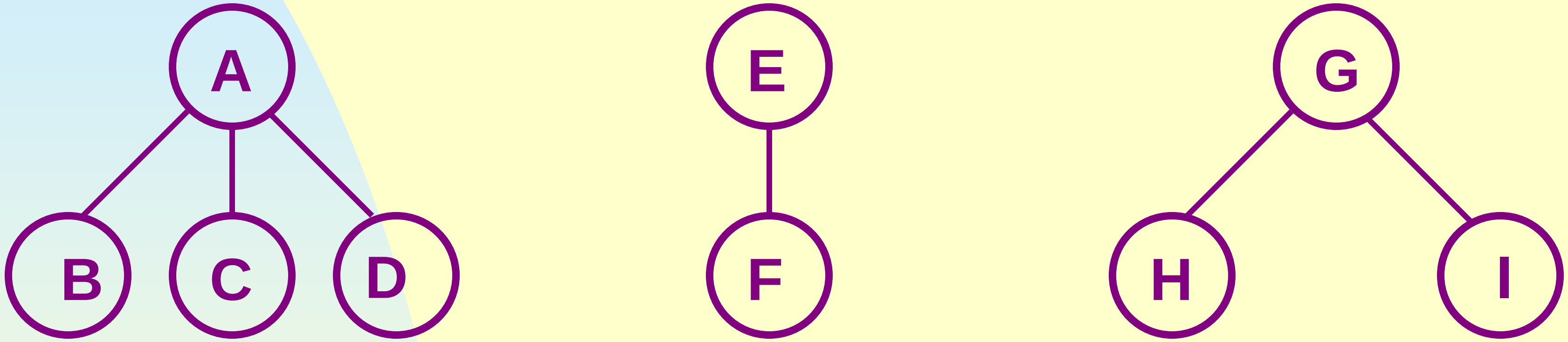


Fig. 5.34: Three-tree forest

5.9.1 Transforming a Forest into a Binary Tree

Definition: If $T_1,\ldots, T_n$ is a forest of trees, then the binary tree corresponding to it , denoted by $B(T_1,\ldots, T_n)$,

(1) is empty if n=0

(2) has root equal to root(T_1); has left subtree equal to $B(T_{11},\ldots, T_{1m})$, where $T_{11},\ldots, T_{1m}$ are the subtrees of root(T_1); and has right subtree $B(T_2,\ldots, T_n)$.

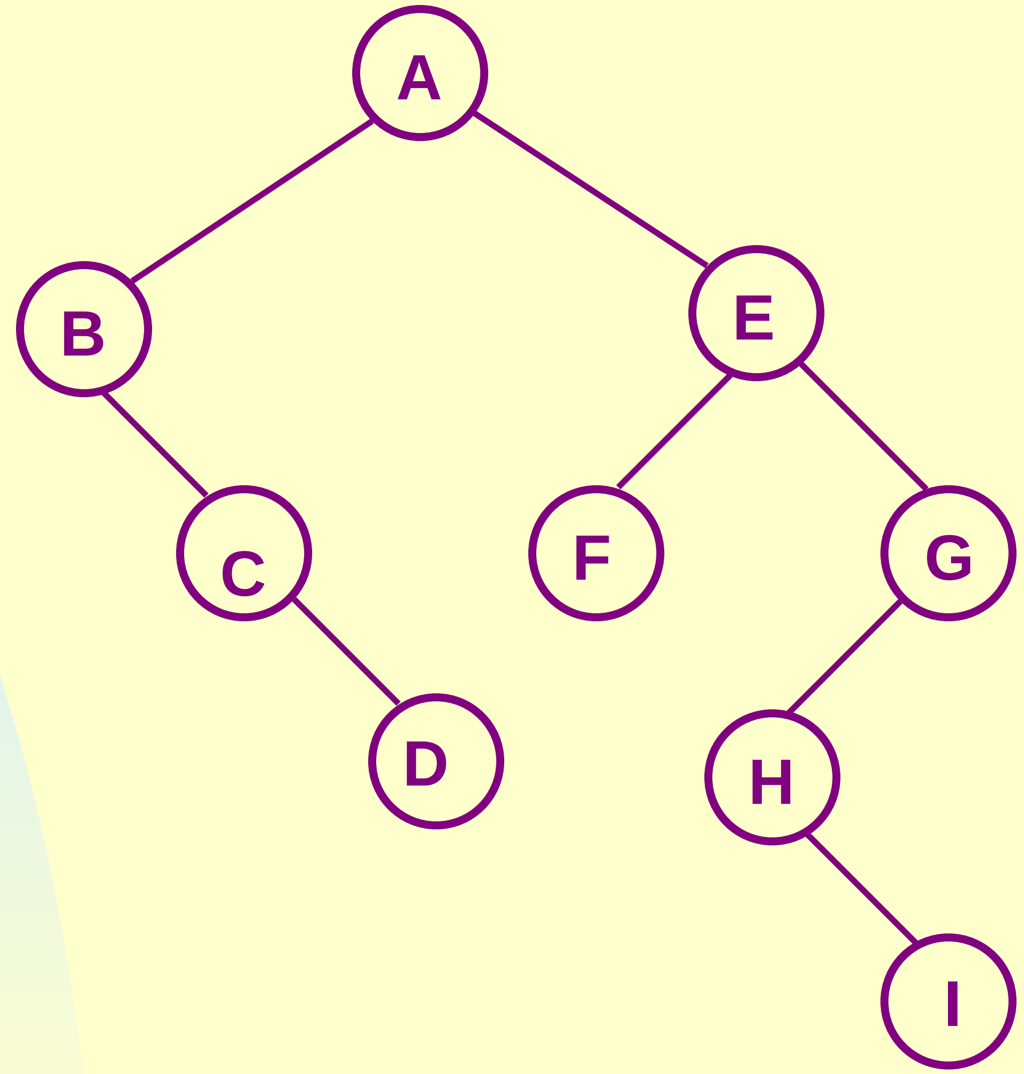


Fig. 5.35: Binary tree representation of forest of Fig.5.34

5.9.2 Forest Traversals

Let T be the corresponding binary tree of a forest F.

Visiting the nodes of F in forest preorder is defined as:

(1) If F is empty then return.

(2) Visit the root of the first tree of F.

(3) Traverse the subtrees of the first tree in forest preorder.

(4) Traverse the remaining trees of F in forest preorder.

Preorder traversal of T is equivalent to visiting the nodes of F in forest preorder

Visiting the nodes of F in forest inorder is defined as:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree in forest inorder.

(3) Visit the root of the first tree of F.

(4) Traverse the remaining trees of F in forest inorder.

Inorder traversal of T is equivalent to visiting the nodes of F in forest inorder

Visiting the nodes of F in forest postorder is defined as:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree in forest postorder.

(3) Traverse the remaining trees of F in forest postorder.

(4) Visit the root of the first tree of F.

There is no natural analog for postorder traversal of the corresponding binary tree of a forest.

In level-order traversal of F, nodes are visited by level, beginning with the roots of each trees in F. Within each level, from left to right.

Exercises: P304-3.

5.10 Representation of disjoint sets

5.10.1 Introduction

Assume:

- Elements of the sets are the numbers 0, 1, 2, …, n-1 (might be thought as indices).
- For any two sets S_i, S_j, $i \neq j$, $S_i \cap S_j = \varnothing$.

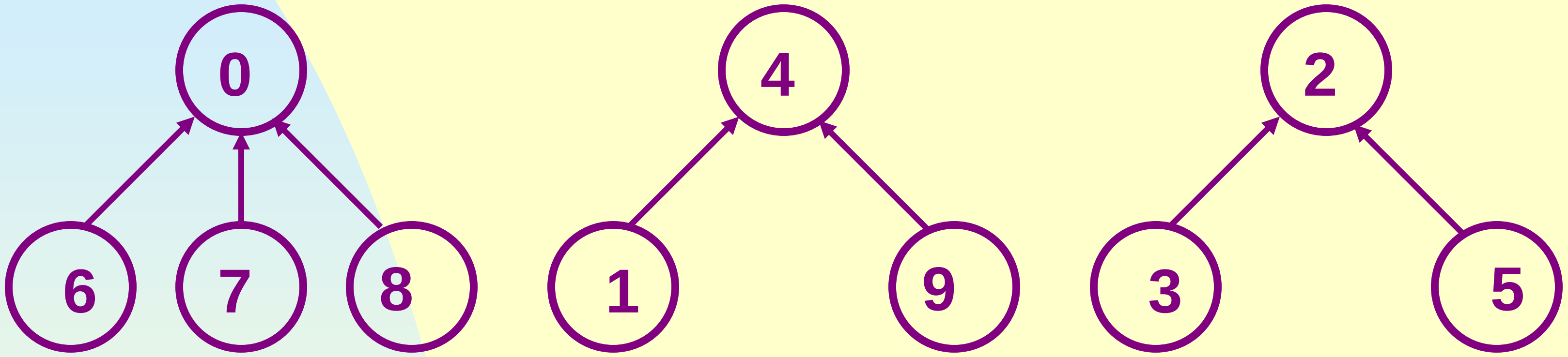
Operations:

(1) Disjoint set union $S_i \cup S_j$.

(2) Find(i)---find the set containing i.

The sets can be represented by trees:

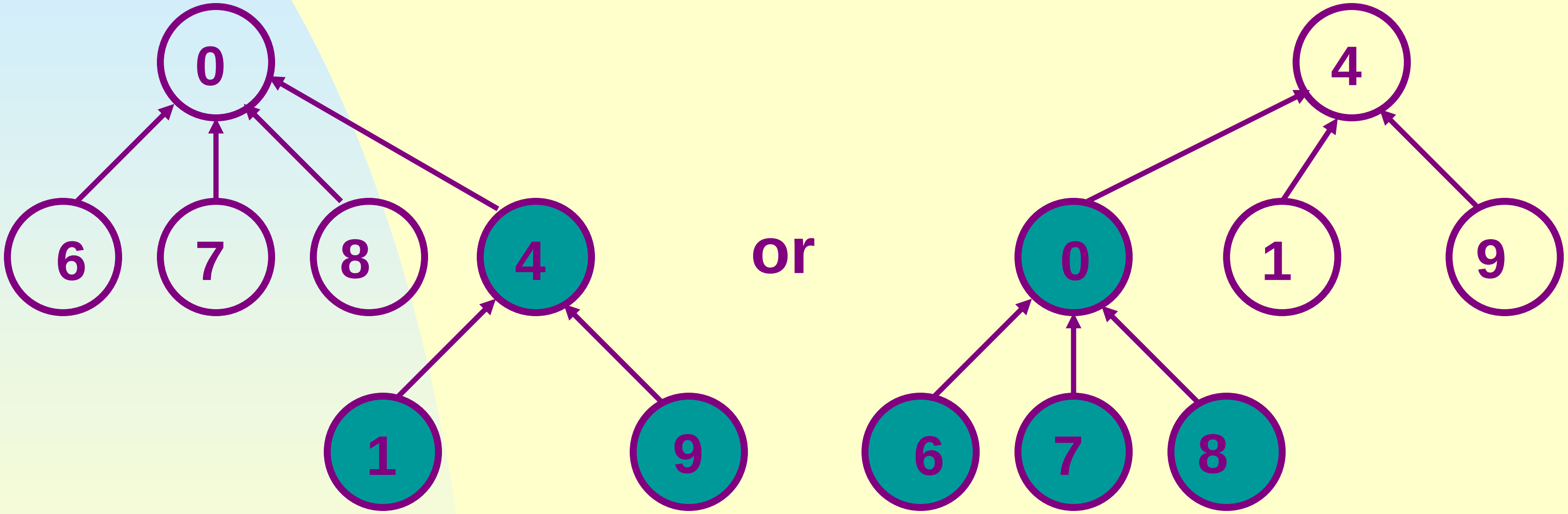
$S_1=\{0,6,7,8\}$ $S_2=\{1,4,9\}$ $S_3=\{2,3,5\}$



The nodes are linked **from the children to the parent.**

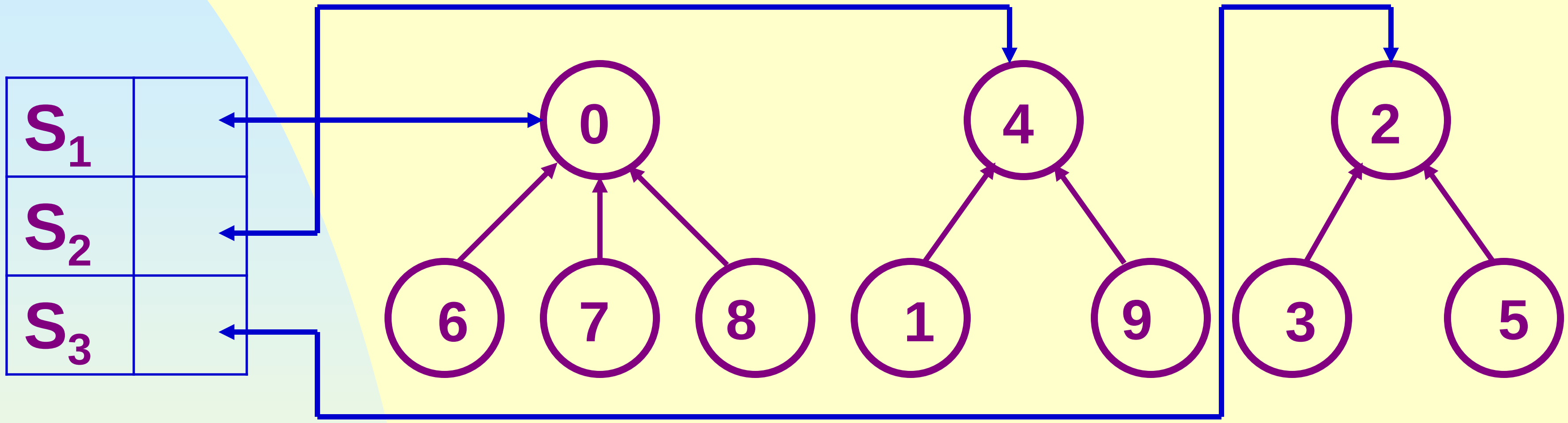
5.10.2 Union and Find Operations

To do $S_1 \cup S_2$, simply make one of the trees a subtree of the other:



$S_1 \cup S_2$

To find the root of a set from its name, we can keep a pointer to the root with each set name.



Here we ignore the actual set names, just identify sets by their roots. The transition to set names is easy.

Since the set elements are numbered 0,1,…,n-1, we represent the tree nodes using an array parent[n]**. The** parent[i] **contains the parent pointer of node i (element i as well).**

The root node has a parent of –1.

The following is the array representation of S_1, S_2, and S_3:

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

```
class Sets {
public:
    // Set operations
    …
private:
    int  *parent;
    int  n; // number of set elements
};

Sets::Sets (int numberOfElements)
{
    if (numberOfElements < 2) throw “Must have at least 2 
elements.”;
    n=numberOfElements;
    parent=new int[n];
    fill(parent, parent+n, -1);
}
```

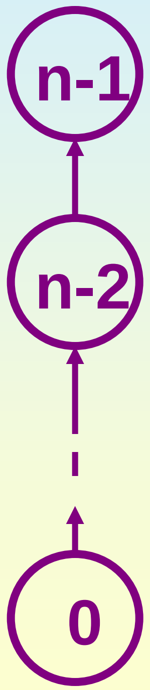
Now we have simple algorithm for union and find.

```
void Sets::SimpleUnion (int i, int j)
{ // Replace the disjoint sets with roots i and j, i!=j with their
  // union
    parent[i] = j;
}

int Sets::SimpleFind (int i)
{ //find the root of the tree containing element i.
    while (parent[i]>=0) i=parent[i];
    return i;
}
```

Analysis of SimpleUnion and SimpleFind:

Given $S_i=\{i\}$, **$0 \le i < n$, do**

Union(0,1), Union(1,2), …, Union(n-2,n-1), Find(0), Find(1),…, Find(n-1) **get**



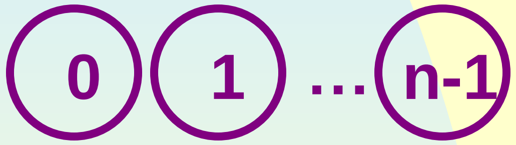
n-1 unions take O(n)

n finds take $O(\sum_{i=1}^{n} i) = O(n^2)$.

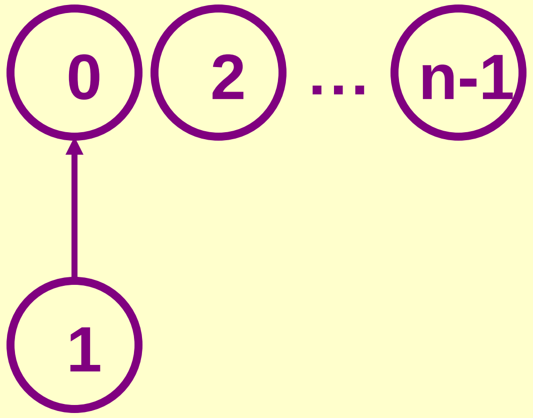
Performance not very good: avoid creating degenerate trees to improve.

Definition [Weighting rule for Union(i,j)]: If the number of nodes in the tree with root i is less than that in the tree with root j, then make j the parent of i; otherwise make i the parent of j.

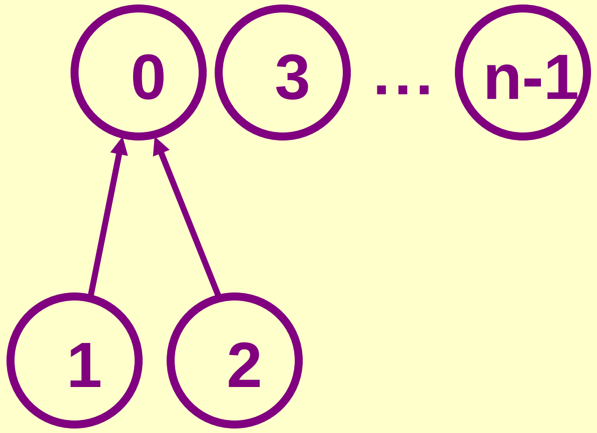
For instance:



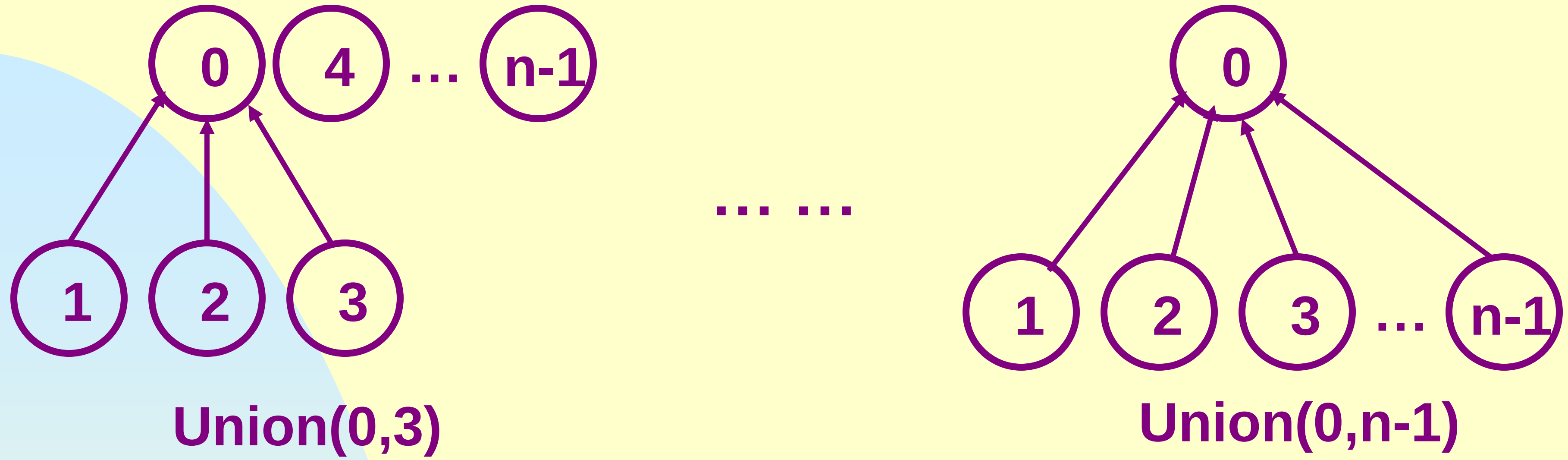
initial



Union(0,1)



Union(0,2)



We need a count field to know how many nodes in each tree, and it can be maintained in the parent field of the root as a negative number.

Since all nodes other than the roots of trees have a non-negative number in the parent field.

```
void Sets::WeightedUnion (int i, int j)
{ // Union sets with roots i and j, i≠j, using the weighting rule
  // parent[i] = - count[i] and parent[j] = - count[j]
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) {   // i has fewer nodes
        parent[i] = j;
        parent[j] = temp;
     }
    else {     // j has fewer nodes or
               // i and j have the same number of nodes
        parent[j] = i;
        parent[i] = temp;
    }
}
```

Analysis of WeightedUnion and SimpleFind:

The time of WeightedUnion is O(1).

The maximum time to perform a find is determined by:

Lemma 5.5: Assume we start with a forest of one node trees. Let T be a tree with m nodes created as a result of a sequence of weighted unions. The height of T is no more than $\lfloor \log_2 m \rfloor + 1$.

Proof by induction:

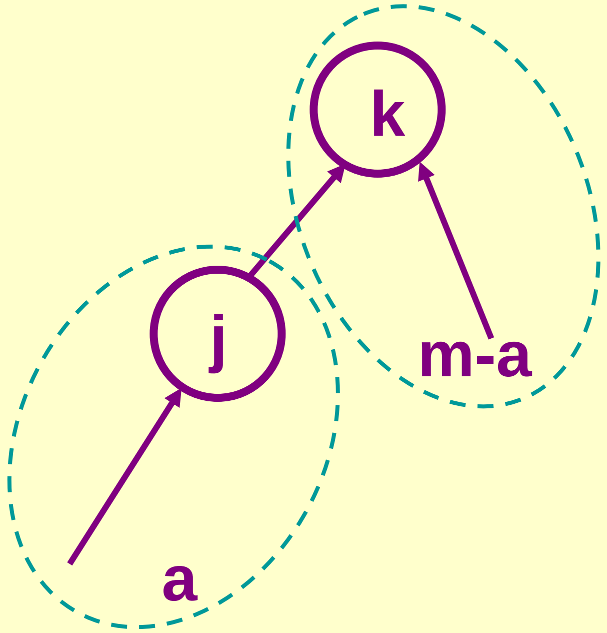
$m = 1$, it is true.

Assume it is true for all trees with $i \leq m-1$ nodes.

For $i = m$, let T be a tree with m nodes created by WeightedUnion.

Consider the last union performed, Union(k, j).

Let a be the number of nodes in tree j and m-a that in tree k. without loss of generality, assume $1 \leq a \leq m/2$. Then the height of T is either the same as that of k or is 1 + that of j.



$m\text{-}a \geq m/2 \geq a$

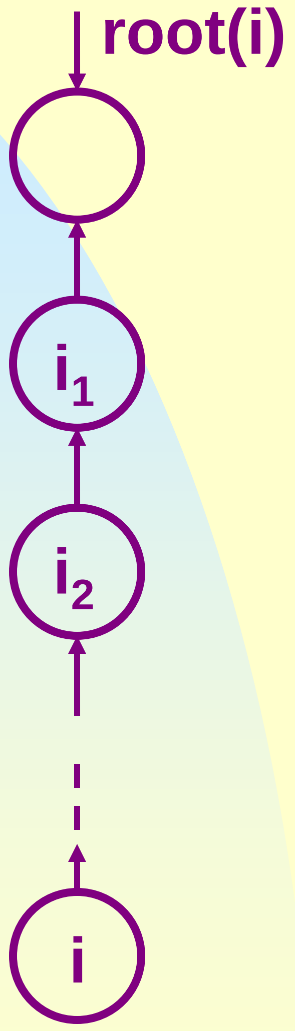
If the former is the case, the height of T $\leq \lfloor \log_2 (m-a) \rfloor + 1 \leq \lfloor \log_2 m \rfloor + 1$.

If the latter is the case, the height of T $\leq \lfloor \log_2 a \rfloor + 2 \leq \lfloor \log_2 m/2 \rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$.

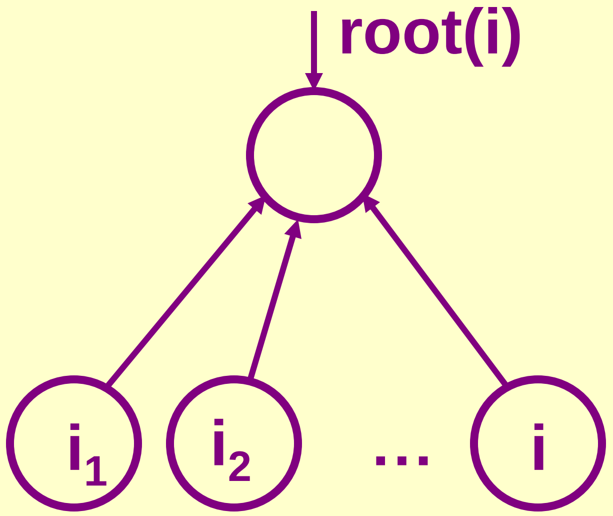
The time to process a find is at most O(log n) in a tree of n nodes.

Further improvement in the find algorithm.

Definition [Collapsing rule] : If j is a node on the path from i to its root and parent[i] ≠ root(i), then set parent[j] to root(i).



➔



$j=i,\ldots, i_2 , i_1$

```
int Sets::CollapsingFind (int i )
{ // Find the root of tree containing element i. Use the
  // collapsing rule to collapse all nodes from i to the root.
    for (int r = i; parent[r] >= 0; r = parent[r]); // find the root of i
    while ( i != r )  {
         int s = parent[i];
         parent[i] = r;
         i = s;
    }
    return r;
}
```

The worst-case complexity of processing a sequence of unions and finds using WeightedUnion and CollapsingFind is stated in Lemma5.6.

First, a very slow growing function

$\alpha(p,q)=\min\{z\geq 1 \mid A(z, \lfloor p/q \rfloor)>\log_2 q\}$, $p \geq q \geq 1$

And Ackermann's function A(i,j) is defined as:

$A(1, j) = 2^j$	for $j \geq 1$
$A(i, 1) = A(i-1, 2)$	for $i \geq 2$
$A(i, j) = A(i-1, A(i, j-1))$	for $i ,j \geq 2$

$A(1, 2) = 2^2$

$A(3,1) = A(2, 2) = A(1, A(2, 1)) = A(1, A(1, 2))$

$= 2^{2^{2}} = 16$

$A(4,1) = A(3, 2) = A(2, A(3, 1)) = A(2, 16)$

$= A(1, A(2, 15)) = 2^{A(2, 15)}$

$= 2^{2^{A(2, 14)}} = \dots = 2^{\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\} 16}$ (very big !)

So for any practical p and q, $\alpha(p,q) \leq 4$.

Lemma 5.6 [Tarjan and Van Leeuwen]: Assume we start with a forest of trees, each having one node. Let T(f, u) be the maximum time required to process any intermixed sequence of f finds and u unions. Assume $u \geq n/2$, then

$$k_1(n+f\,\alpha(f+n,n)) \leq T(f, u) \leq k_2\,(n+f\,\alpha(f+n,n))$$

For some positive constants k_1 and k_2.

Note T(f, u) is not linear even though $\alpha(f+n,n)$ grows very slow.

The **space** requirements are one node for each element.

5.10.3 Application to Equivalent Classes

equivalence classes $\Leftrightarrow$ disjoint sets

Initially, parent[i] = -1, $0 \le i \le n-1$.

To process $i \equiv j$,

Let x = find(i), y = find(j) --- 2 finds

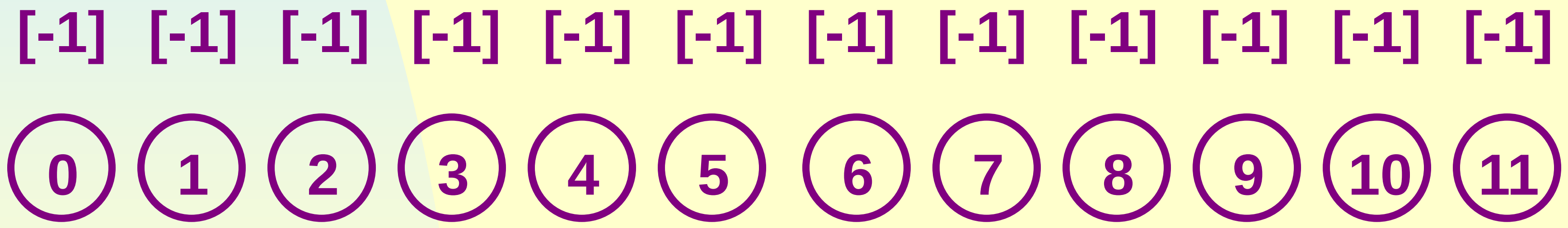
If $x \neq y$ then union(x, y) --- at most 1 union

Thus if we have n elements and m equivalence pairs, we needs 2m finds and min {n-1, m} unions. The total time is $O(n+2m\,\alpha(n+2m,n))$.

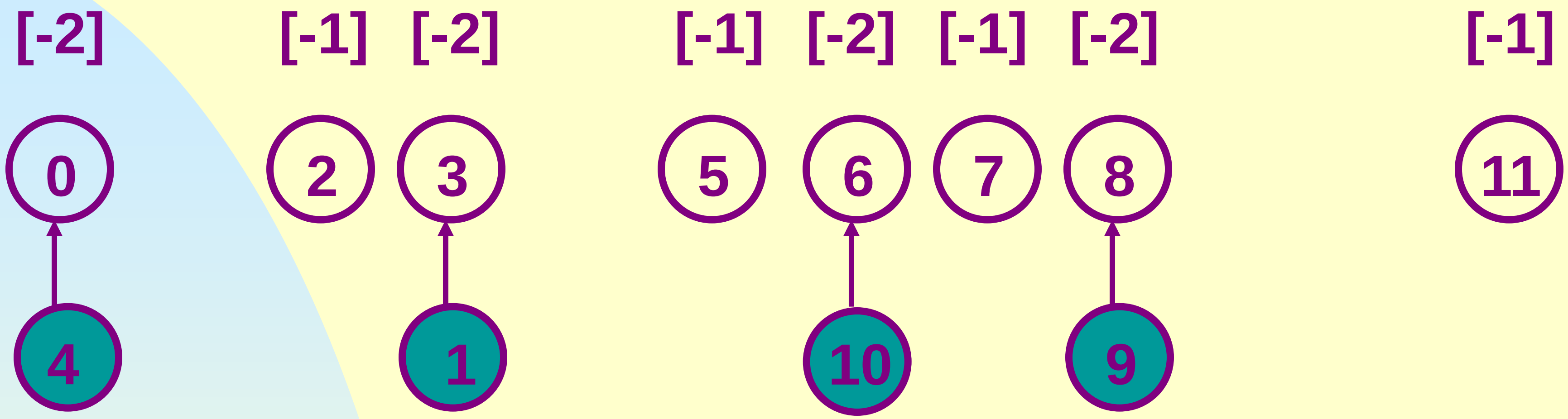
Example:

n = 12, process equivalence pairs:

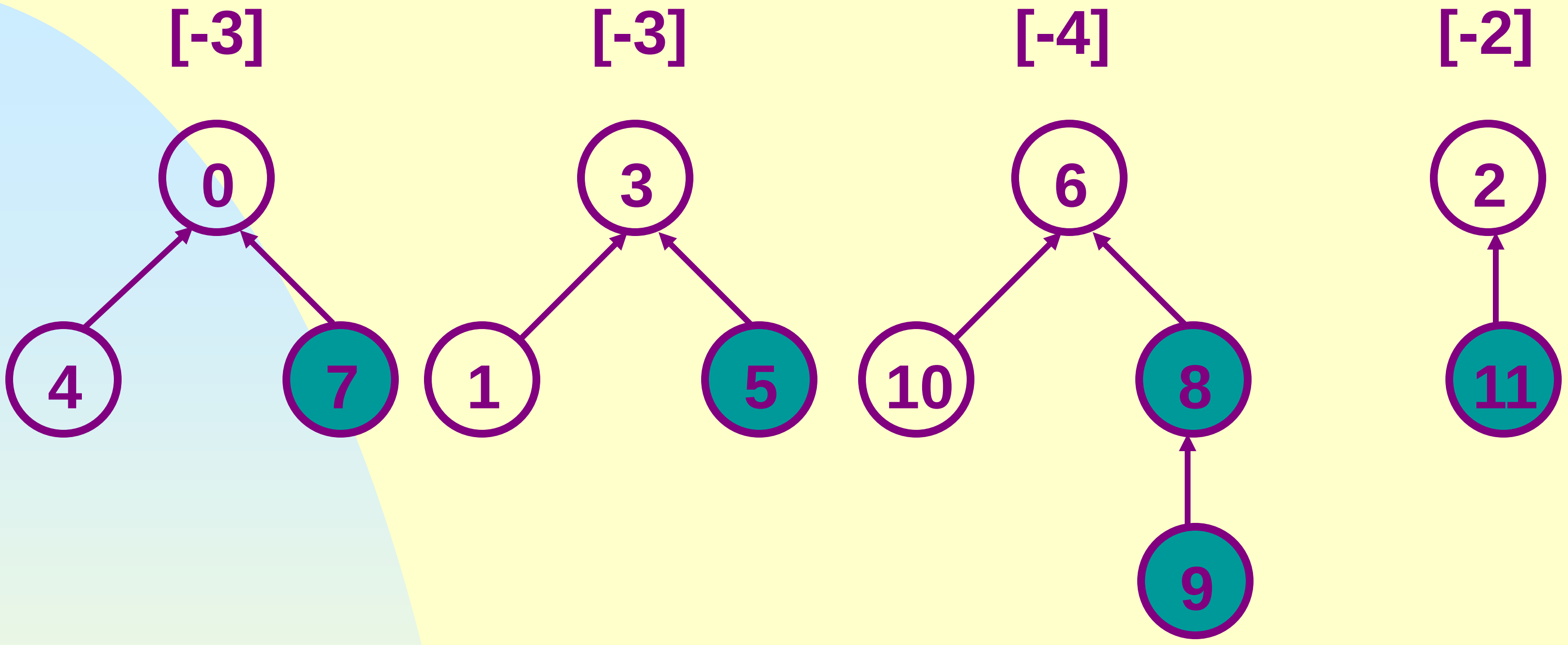
0 ≡ 4, 3 ≡ 1, 6 ≡10, 8 ≡ 9, 7≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, 11 ≡ 0



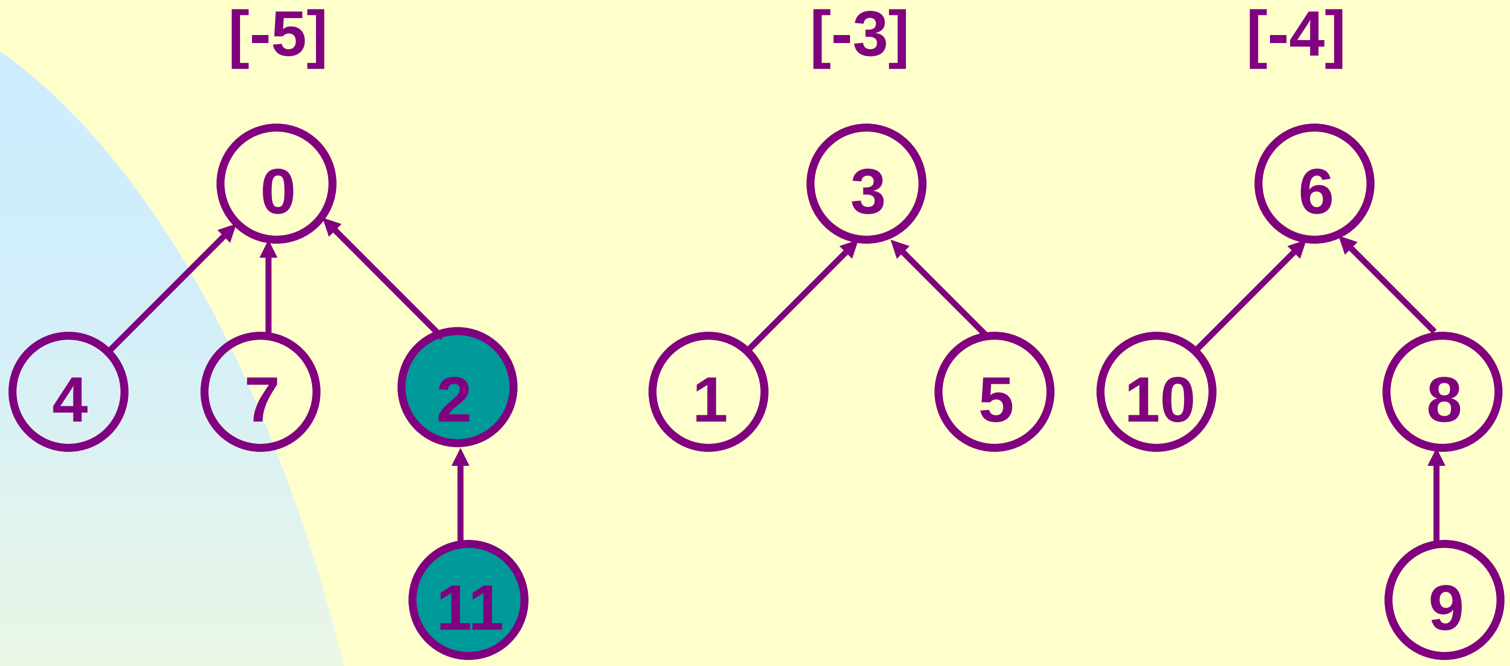
(a) Initial trees



(b) After processing 0 ≡ 4, 3 ≡ 1, 6 ≡10, and 8 ≡ 9



(c) After processing 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, and 2 ≡ 11



(d) After processing 11 ≡ 0

Exercises: P316-3