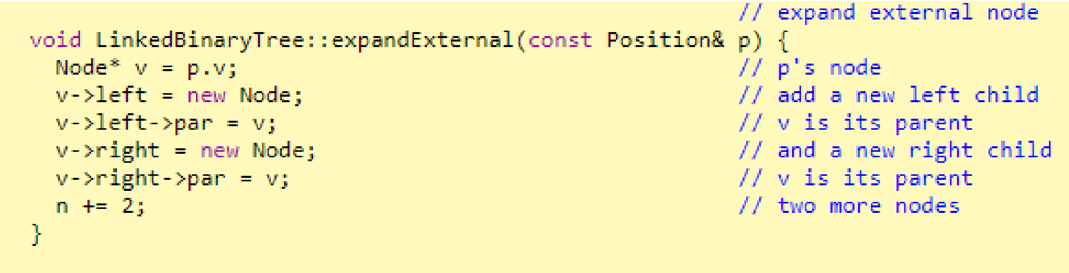
# BITS F232: FOUNDATIONS OF DATA STRUCTURES & ALGORITHMS (1$^{ST}$ SEMESTER 2023-24) TREE ADT CONTINUED…

Chittaranjan Hota, PhD
Sr. Professor of Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

# RECAP

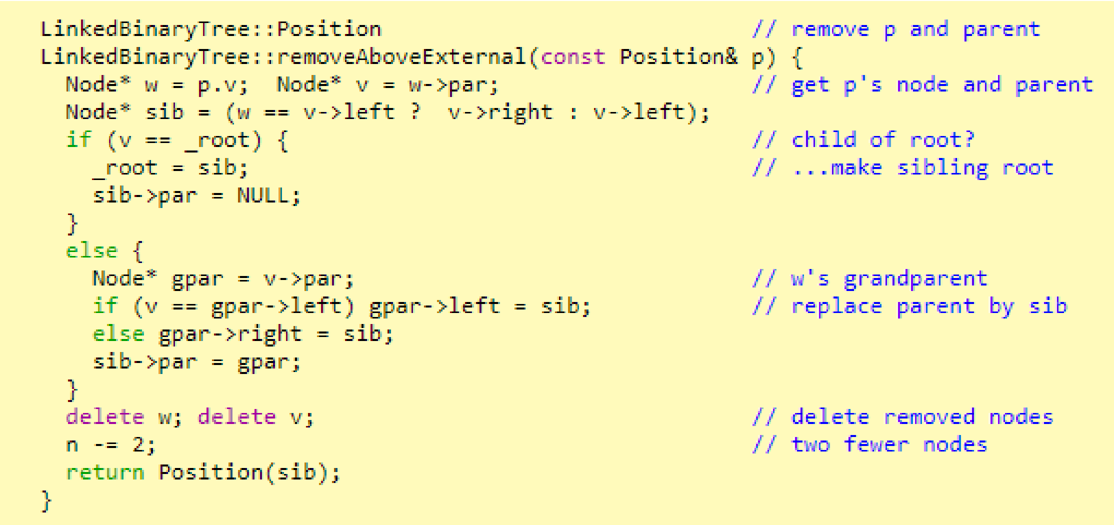
# BINARY TREE UPDATE FUNCTIONS

## expandExternal(const Position& p)



```
                                                      // expand external node
void LinkedBinaryTree::expandExternal(const Position& p) {
  Node* v = p.v;                                      // p's node
  v->left = new Node;                                 // add a new left child
  v->left->par = v;                                   // v is its parent
  v->right = new Node;                                // and a new right child
  v->right->par = v;                                  // v is its parent
  n += 2;                                             // two more nodes
}
```

## removeAboveExternal (const Position& p)



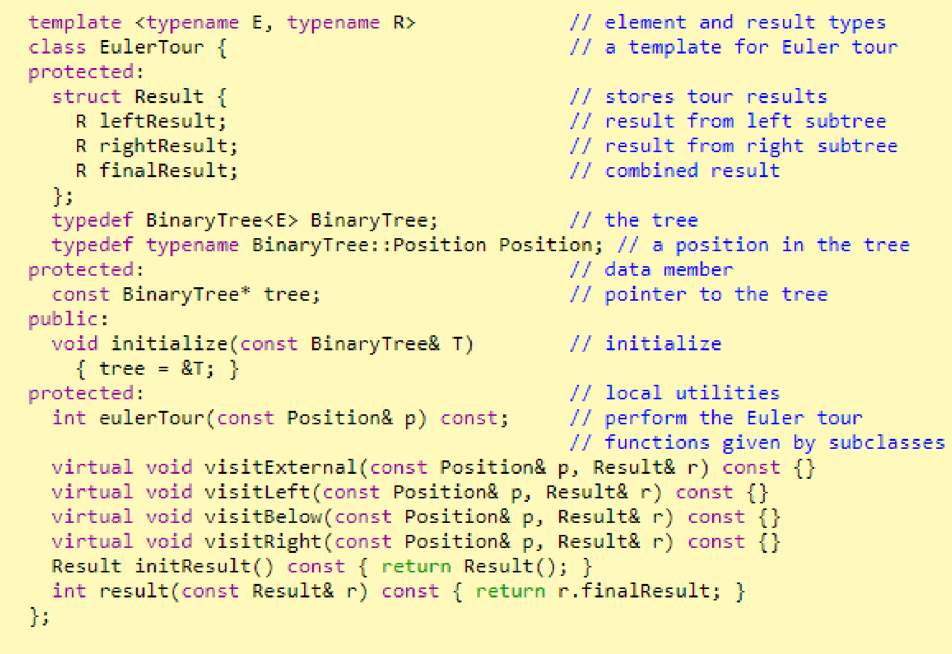
```
LinkedBinaryTree::Position                            // remove p and parent
LinkedBinaryTree::removeAboveExternal(const Position& p) {
  Node* w = p.v;  Node* v = w->par;                   // get p's node and parent
  Node* sib = (w == v->left ?  v->right : v->left);
  if (v == _root) {                                   // child of root?
    _root = sib;                                      // ...make sibling root
    sib->par = NULL;
  }
  else {
    Node* gpar = v->par;                              // w's grandparent
    if (v == gpar->left) gpar->left = sib;            // replace parent by sib
    else gpar->right = sib;
    sib->par = gpar;
  }
  delete w; delete v;                                 // delete removed nodes
  n -= 2;                                             // two fewer nodes
  return Position(sib);
}
```

# BINARY SEARCH TREES

**More later …**

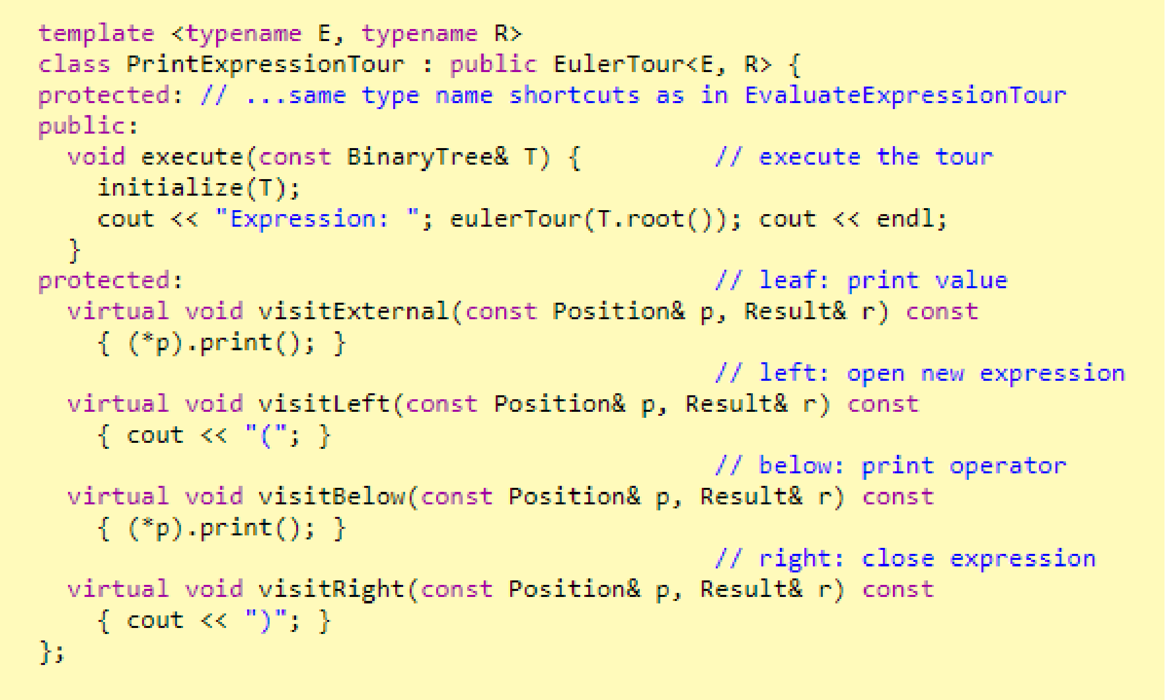
# THE TEMPLATE FUNCTION PATTERN

- The template function pattern describes a generic computation method that can be tuned for a particular application by redefining certain steps.



```
template <typename E, typename R>              // element and result types
class EulerTour {                              // a template for Euler tour
protected:
  struct Result {                              // stores tour results
    R leftResult;                              // result from left subtree
    R rightResult;                             // result from right subtree
    R finalResult;                             // combined result
  };
  typedef BinaryTree<E> BinaryTree;            // the tree
  typedef typename BinaryTree::Position Position; // a position in the tree
protected:                                     // data member
  const BinaryTree* tree;                      // pointer to the tree
public:
  void initialize(const BinaryTree& T)         // initialize
    { tree = &T; }
protected:                                     // local utilities
  int eulerTour(const Position& p) const;      // perform the Euler tour
                                               // functions given by subclasses
  virtual void visitExternal(const Position& p, Result& r) const {}
  virtual void visitLeft(const Position& p, Result& r) const {}
  virtual void visitBelow(const Position& p, Result& r) const {}
  virtual void visitRight(const Position& p, Result& r) const {}
  Result initResult() const { return Result(); }
  int result(const Result& r) const { return r.finalResult; }
};
```

(Class EulerTour defining a generic Euler tour of a binary tree. It realizes template function pattern and must be specialized for use)

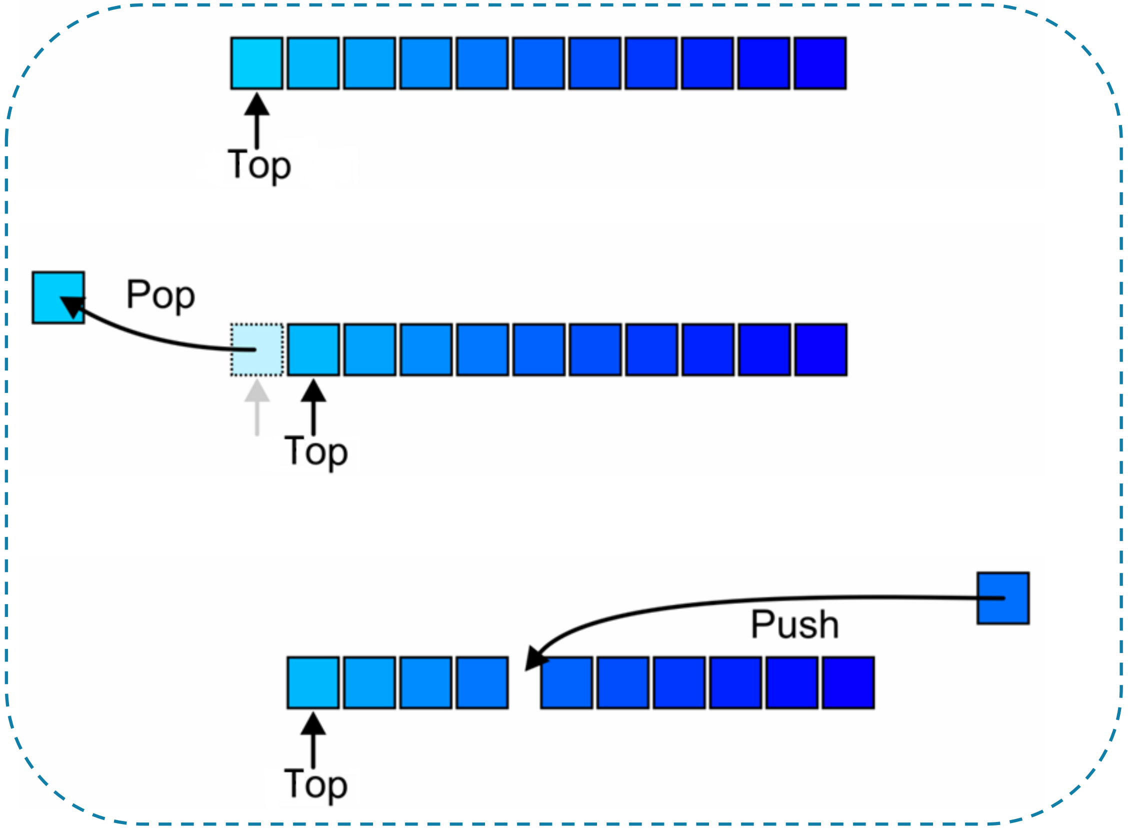


```
template <typename E, typename R>
class PrintExpressionTour : public EulerTour<E, R> {
protected: // ...same type name shortcuts as in EvaluateExpressionTour
public:
  void execute(const BinaryTree& T) {          // execute the tour
    initialize(T);
    cout << "Expression: "; eulerTour(T.root()); cout << endl;
  }
protected:                                     // leaf: print value
  virtual void visitExternal(const Position& p, Result& r) const
    { (*p).print(); }
                                               // left: open new expression
  virtual void visitLeft(const Position& p, Result& r) const
    { cout << "("; }
                                               // below: print operator
  virtual void visitBelow(const Position& p, Result& r) const
    { (*p).print(); }
                                               // right: close expression
  virtual void visitRight(const Position& p, Result& r) const
    { cout << ")"; }
};
```

(A derived class, called PrintExpressionTour that prints the arithmetic expression)

# PRIORITY QUEUES

- We have discussed Abstract Lists with explicit linear orders (Arrays, Linked lists etc.)
- We also discussed containers with restricted operations (Stacks, Queues etc.)
- Priority queues will ensure implicit linear ordering amongst the objects.
- Queues: Order is decided by FCFS
- Priority Queues: Objects have a priority associated with them and we use this to remove (pop out) either the highest or lowest priority object depending on the applications need.



# LEXICOGRAPHICAL PRIORITY

Ex1: What is priority boarding on aircrafts?

Ex2: how did you decide BITS campus to choose from several institutions?

Priority may also depend on multiple variables:

- Two values specify a priority: $(a, b)$
- A pair $(a, b)$ has higher priority than $(c, d)$ if:
  - $a < c$, or
  - $a = c$ and $b < d$

Mathematical concept of total order relation $\leq$

- Reflexive property: $\boldsymbol{x} \leq \boldsymbol{x}$
- Antisymmetric property: $\boldsymbol{x} \leq \boldsymbol{y} \wedge \boldsymbol{y} \leq \boldsymbol{x} \Rightarrow \boldsymbol{x} = \boldsymbol{y}$
- Transitive property: $\boldsymbol{x} \leq \boldsymbol{y} \wedge \boldsymbol{y} \leq \boldsymbol{z} \Rightarrow \boldsymbol{x} \leq \boldsymbol{z}$

For example,

- (5, 19), (13, 1), (13, 24), and (15, 0) all have *higher* priority than (15, 7)