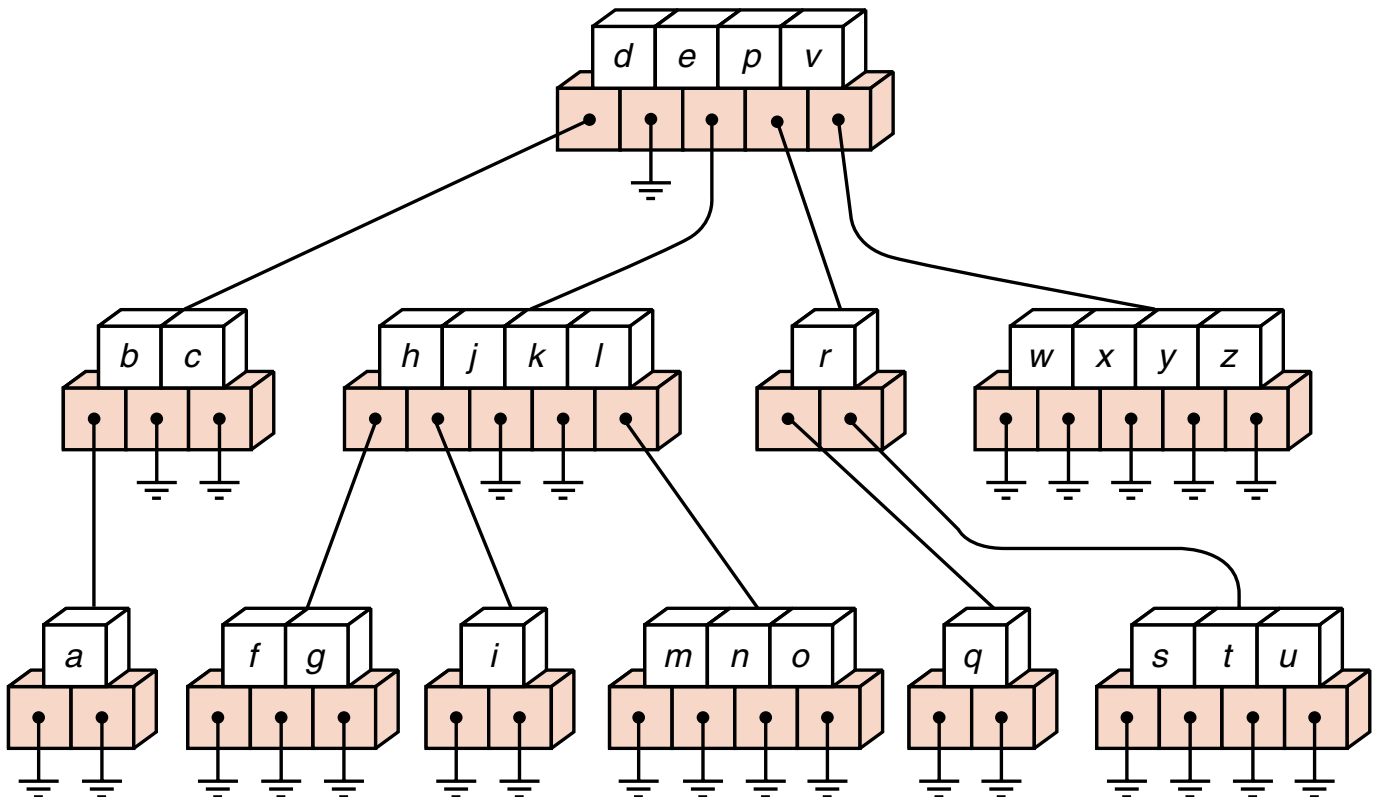# Multiway Search Trees

- An *m-way search tree* is a tree in which, for some integer m called the **order** of the tree, each node has at most m children.

- If $k \leq m$ is the number of children, then the node contains exactly $k - 1$ keys, which partition all the keys into k subsets consisting of all the keys less than the first key in the node, all the keys between a pair of keys in the node, and all keys greater than the largest key in the node.

Balanced Multiway Trees (B-Trees)

DEFINITION A ***B-tree of order*** m is an m-way search tree in which

1. All leaves are on the same level.

2. All internal nodes except the root have at most m non-empty children, and at least $\lceil m/2 \rceil$ nonempty children.

3. The number of keys in each internal node is one less than the number of its nonempty children, and these keys partition the keys in the children in the fashion of a search tree.

4. The root has at most m children, but may have as few as 2 if it is not a leaf, or none if the tree consists of the root alone.
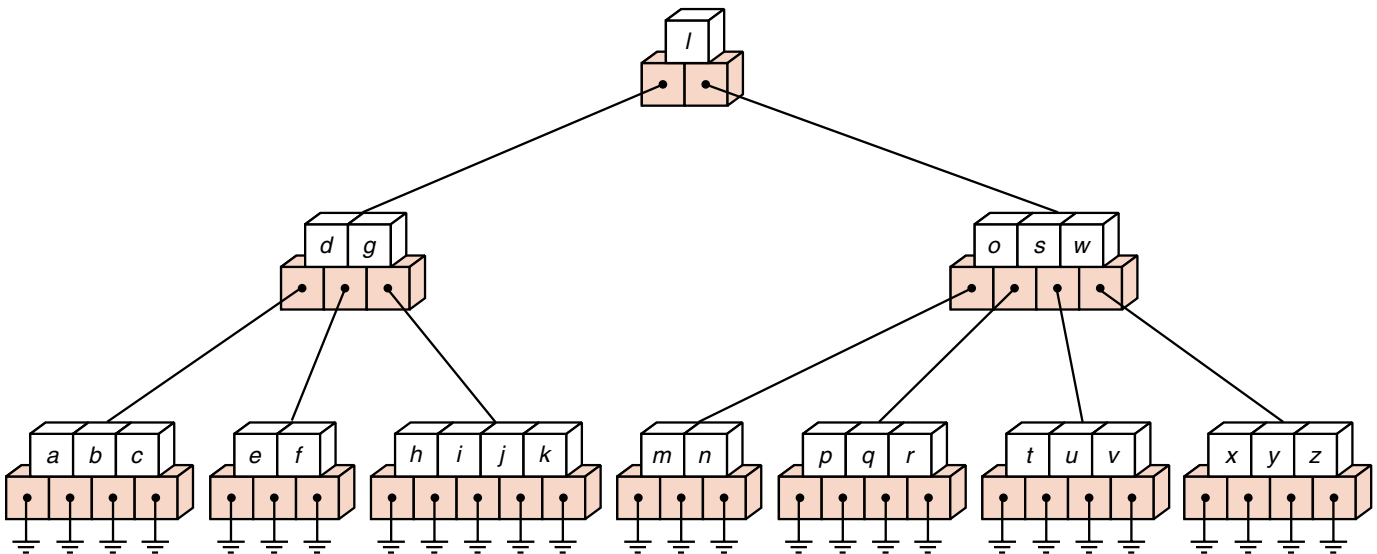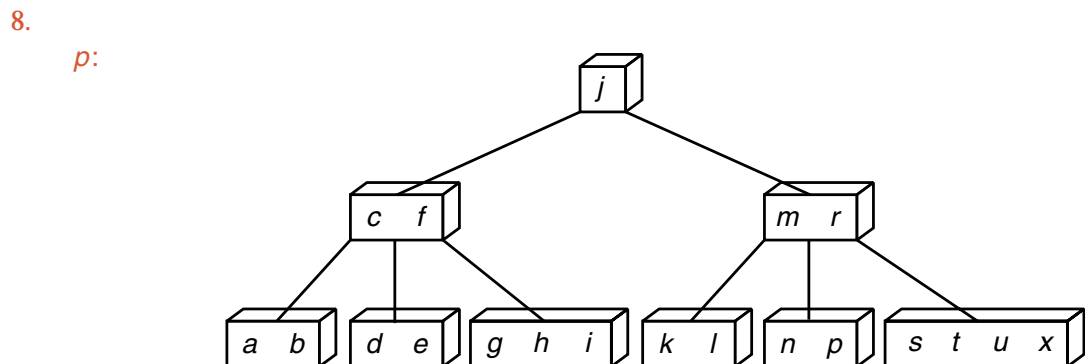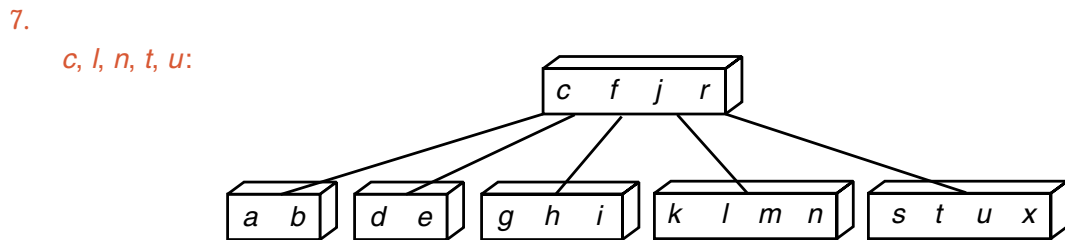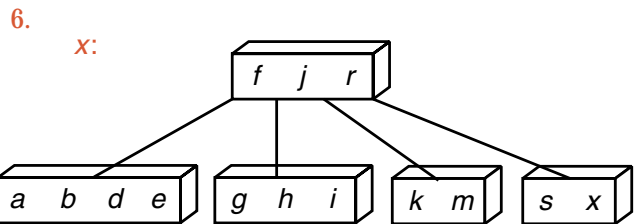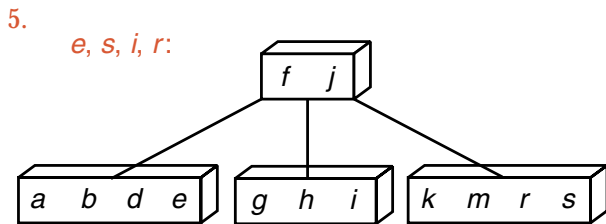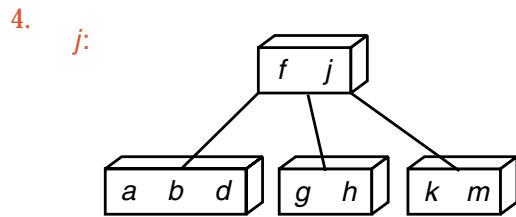
Insertion into a B-Tree

In contrast to binary search trees, B-trees are not allowed to grow at their leaves; instead, they are forced to grow at the root. General insertion method:

1. Search the tree for the new key. This search (if the key is truly new) will terminate in failure at a leaf.

2. Insert the new key into to the leaf node. If the node was not previously full, then the insertion is finished.

3. When a key is added to a full node, then the node splits into two nodes, side by side on the same level, except that the median key is not put into either of the two new nodes.

4. When a node splits, move up one level, insert the median key into this parent node, and repeat the splitting process if necessary.

5. When a key is added to a full root, then the root splits in two and the median key sent upward becomes a new root. This is the only time when the B-tree grows in height.

Growth of a B-Tree

1. *a, g, f, b*:

```
| a   b   f   g |
```

2. *k*:

```
        f
       / \
   a b   g k
```

3. *d, h, m*:

```
        f
       / \
  a b d   g h k m
```

4. *j*:

```
        f   j
       / |  \
  a b d  g h  k m
```

5. *e, s, i, r*:

```
         f   j
        /  |   \
 a b d e  g h i  k m r s
```

6. *x*:

```
          f   j   r
        /  |   |   \
 a b d e  g h i  k m  s x
```

7. *c, l, n, t, u*:

```
              c   f   j   r
            / |   |   |   \
       a b  d e  g h i  k l m n  s t u x
```

8. *p*:

```
                     j
                   /   \
             c   f       m   r
            / | \        / | \
      a b  d e  g h i  k l  n p  s t u x
```
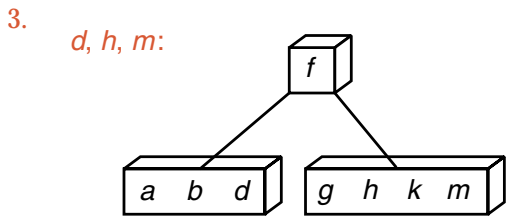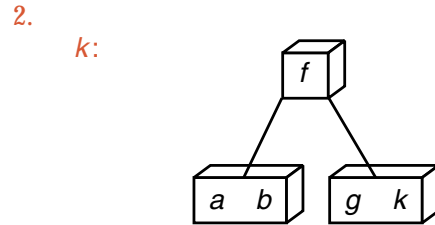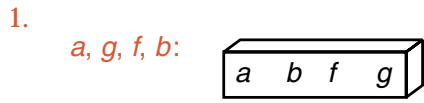
B-Tree Declarations in C++

We add the order as a second template parameter. For example,
B_tree<**int**, 5> sample_tree; declares sample_tree as a B_tree of order
5 that holds integer records.

B-tree class declaration:

```
template <class Record, int order>
class B_tree {
public:                          //    Add public methods.
private:                         //    data members
   B_node<Record, order> *root;
                                 //    Add private auxiliary functions here.
};
```

Node declaration:

```
template <class Record, int order>
struct B_node {
//    data members:
   int count;
   Record data[order − 1];
   B_node<Record, order> *branch[order];
//    constructor:
   B_node();
};
```

Conventions:

- count gives the number of records in the B_node.

- If count is nonzero then the node has count $+ 1$ children.

- branch[0] points to the subtree containing all records with keys less than that in data[0].

- For $1 \leq$ position $\leq$ count $- 1$, branch[position] points to the subtree with keys strictly between those in the subtrees pointed to by data[position $- 1$] and data[position].

- branch[count] points to the subtree with keys greater than that of data[count $- 1$].

Searching in a B-Tree

Public method:

```
template <class Record, int order>
Error_code B_tree<Record, order> :: search_tree(Record &target)
/* Post:  If there is an entry in the B-tree whose key matches that in target, the pa-
          rameter target is replaced by the corresponding Record from the B-tree
          and a code of success is returned.  Otherwise a code of not_present
          is returned.
   Uses: recursive_search_tree */
{
   return recursive_search_tree(root, target);
}
```

Recursive function:

```cpp
template <class Record, int order>
Error_code B_tree<Record, order> :: recursive_search_tree(
          B_node<Record, order> *current, Record &target)
/* Pre:   current is either NULL or points to a subtree of the B_tree.
   Post:  If the Key of target is not in the subtree, a code of not_present is
          returned. Otherwise, a code of success is returned and target is set to
          the corresponding Record of the subtree.
   Uses: recursive_search_tree recursively and search_node */

{
   Error_code result = not_present;
   int position;
   if (current != NULL) {
      result = search_node(current, target, position);
      if (result == not_present)
         result =
            recursive_search_tree(current->branch[position], target);
      else
         target = current->data[position];
   }
   return result;
}
```

- This function has been written recursively to exhibit the similarity of its structure to that of the insertion function.

- The recursion is tail recursion and can easily be replaced by iteration.

Searching a Node

This function determines if the target is present in the current node, and, if not, finds which of the count + 1 branches will contain the target key.

```
template <class Record, int order>
Error_code B_tree<Record, order>::search_node(
   B_node<Record, order> *current, const Record &target, int &position)
/* Pre:   current points to a node of a B_tree.
   Post:  If the Key of target is found in *current, then a code of success is returned,
          the parameter position is set to the index of target, and the corresponding
          Record is copied to target.  Otherwise, a code of not_present is returned,
          and position is set to the branch index on which to continue the search.
   Uses:  Methods of class Record.  */
{
   position = 0;
   while (position < current->count && target > current->data[position])
      position++;                      //   Perform a sequential search through the keys.
   if (position < current->count && target ==  current->data[position])
      return success;
   else
      return not_present;
}
```

- For B-trees of large order, this function should be modified to use binary search instead of sequential search.

- Another possibility is to use a linked binary search tree instead of a sequential array of entries for each node.

Insertion: Parameters and push_down

- Insertion is done with recursion in a function called push_down.

- We require that the record new_entry being inserted is not already present in the tree.

The recursive function push_down uses three more output parameters.

- current is the root of the current subtree under consideration.

- If *current splits to accommodate new_entry, push_down returns a code of overflow, and the following come into use:

- The old node *current contains the left half of the entries.

- median gives the median record.

- right_branch points to a new node containing the right half of the former *current.
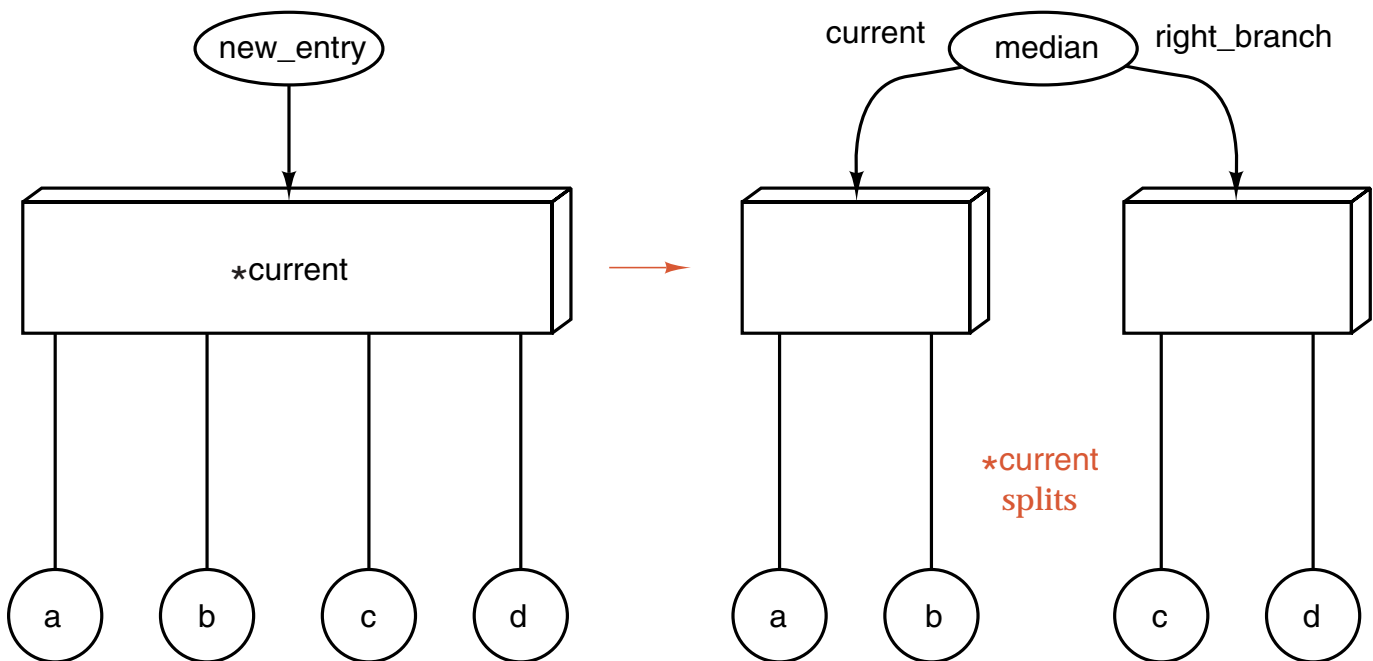
Public Insertion Method

template <**class** Record, **int** order>
Error_code B_tree<Record, order> **::** insert(**const** Record &new_entry)
/* Post: *If the* Key *of* new_entry *is already in the* B_tree, *a code of* duplicate_error
is returned. Otherwise, a code of success *is returned and the* Record
new_entry *is inserted into the B-tree in such a way that the properties of a
B-tree are preserved.*
  Uses: *Methods of* **struct** B_node *and the auxiliary function* push_down. ***/**

```
{
  Record median;
  B_node<Record, order> *right_branch, *new_root;
  Error_code result =
    push_down(root, new_entry, median, right_branch);

  if (result ==  overflow) {        //   The whole tree grows in height.
                                    //   Make a brand new root for the whole B-tree.
    new_root = new B_node<Record, order>;
    new_root->count = 1;
    new_root->data[0] = median;
    new_root->branch[0] = root;
    new_root->branch[1] = right_branch;
    root = new_root;
    result = success;
  }
  return result;
}
```

Recursive Insertion into a Subtree

template <**class** Record, **int** order>
Error_code B_tree<Record, order> :: push_down(
           B_node<Record, order> *current,
           const Record &new_entry,
           Record &median,
           B_node<Record, order> * &right_branch)
/* **Pre:**   current *is either* NULL *or points to a node of a* B_tree.

  Post: *If an entry with a* Key *matching that of* new_entry *is in the subtree to which* current *points, a code of* duplicate_error *is returned. Otherwise,* new_entry *is inserted into the subtree: If this causes the height of the subtree to grow, a code of* overflow *is returned, and the* Record median *is extracted to be reinserted higher in the B-tree, together with the subtree* right_branch *on its right. If the height does not grow, a code of* success *is returned.*

  Uses: *Functions* push_down *(called recursively),* search_node, split_node, *and* push_in. */

```
{
  Error_code result;
  int position;
  if (current ==  NULL) {
            //    Since we cannot insert in an empty tree, the recursion terminates.
    median = new_entry;
    right_branch = NULL;
    result = overflow;
  }
```

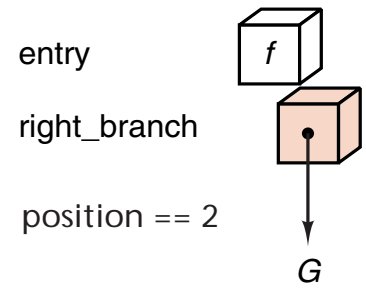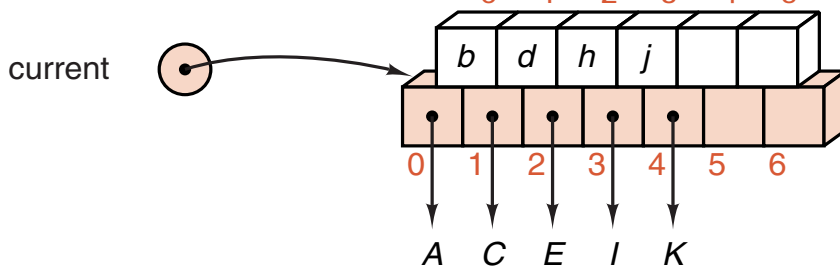# Recursive Insertion, Continued

```
else {                              //    Search the current node.
  if (search_node(current, new_entry, position) ==  success)
    result = duplicate_error;
  else {
    Record extra_entry;
    B_node<Record, order> *extra_branch;
    result = push_down(current->branch[position], new_entry,
                          extra_entry, extra_branch);

    if (result ==  overflow) {
                        //    Record extra_entry now must be added to current
      if (current->count < order − 1) {
        result = success;
        push_in(current, extra_entry, extra_branch, position);
      }

      else split_node(    current, extra_entry, extra_branch, position,
                          right_branch, median);
      //    Record median and its right_branch will go up to a higher node.
    }
  }
}
return result;
}
```
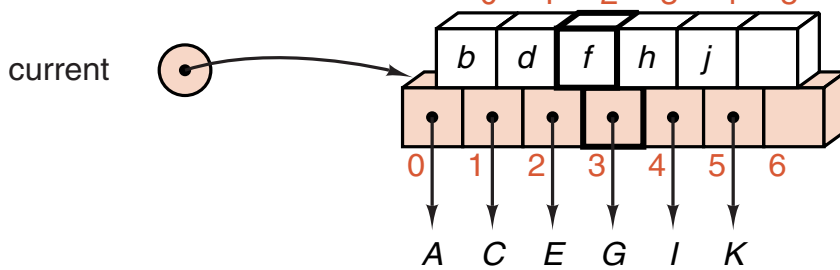
Before:

current

	0	1	2	3	4	5
	b	d	h	j		

branch: 0 1 2 3 4 5 6

A C E I K

entry → f

right_branch

position == 2 → G

After:

current

	0	1	2	3	4	5
	b	d	f	h	j	

branch: 0 1 2 3 4 5 6

A C E G I K

```
template <class Record, int order>
void B_tree<Record, order> ::
    push_in(B_node<Record, order> *current,
            const Record &entry,
            B_node<Record, order> *right_branch,
            int position)
```
/* **Pre:** current *points to a node of a* B_tree. *The node* *current *is not full and* entry
         belongs in *current *at index* position.
   Post: entry *has been inserted along with its right-hand branch* right_branch *into*
         *current *at index* position. */
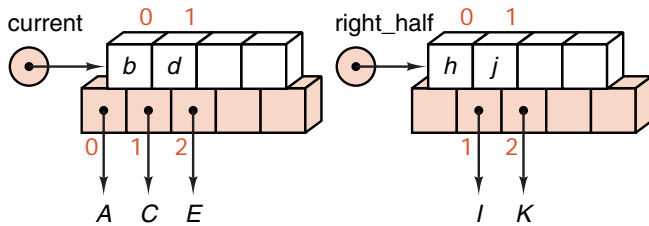```
{
  for (int i = current->count; i > position; i−−) {
                              //   Shift all later data to the right.
    current->data[i] = current->data[i − 1];
    current->branch[i + 1] = current->branch[i];
  }
  current->data[position] = entry;
  current->branch[position + 1] = right_branch;
  current->count++;
}
```
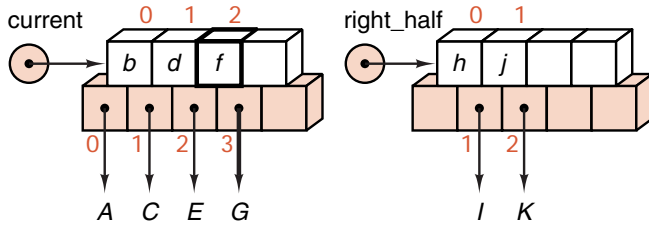
# Example of Splitting a Full Node

**Case 1:** position == 2; order == 5;
(extra_entry belongs in left half.)



mid = 2;
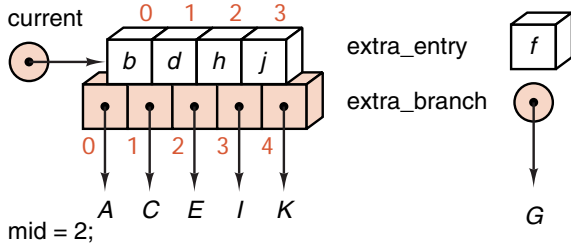
Shift entries right:



Insert extra_entry and extra_branch:



Remove median; move branch:



**Case 2:** position == 3; order == 5;
(extra_entry belongs in right half.)



mid = 3;
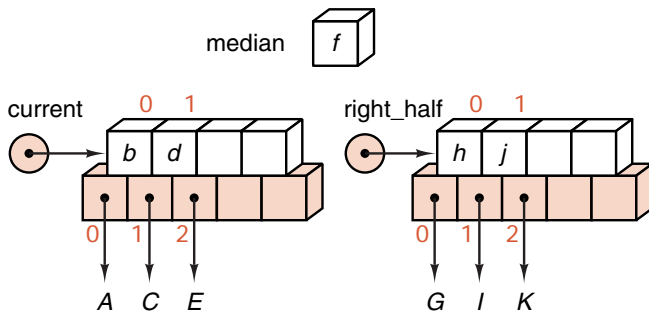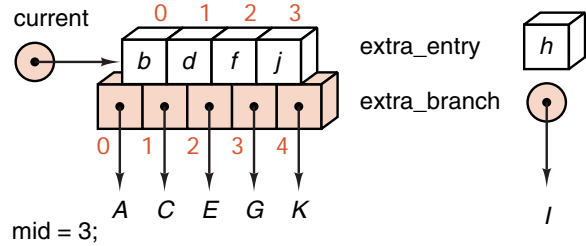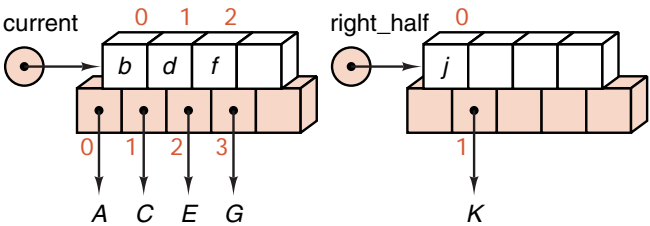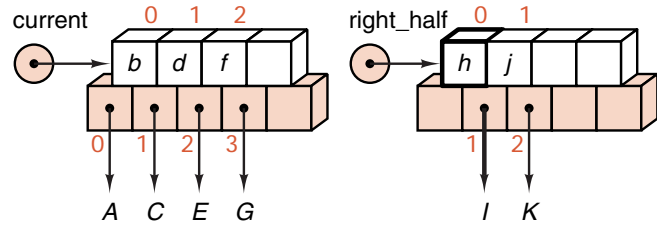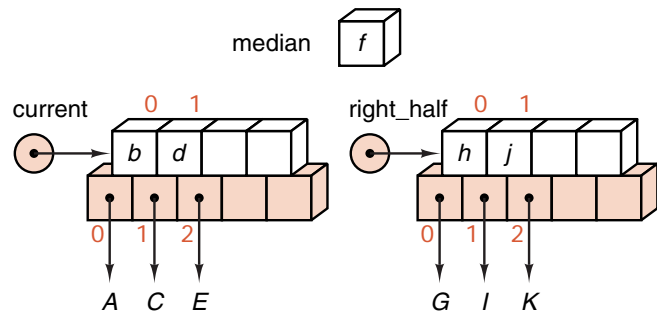
Shift entry right:



Insert extra_entry and extra_branch:



Remove median; move branch:

# Function split_node, Specifications

**template** <**class** Record, **int** order>
**void** B_tree<Record, order> **::** split_node(
   B_node<Record, order> ∗current, **//**    *node to be split*
   **const** Record &extra_entry, **//**   *new entry to insert*
   B_node<Record, order> ∗extra_branch,
               **//**    *subtree on right of* extra_entry
   **int** position,     **//**    *index in node where* extra_entry *goes*
   B_node<Record, order> ∗ &right_half, **//**    *new node for right half of entries*
   Record &median) **//**    *median entry (in neither half)*
**/∗ Pre:**   current *points to a node of a* B_tree*. The node* ∗current *is full, but if there*
         *were room, the record* extra_entry *with its right-hand pointer* extra_branch
         *would belong in* ∗current *at position* position, $0 \leq$ position $<$ order*.*
  **Post:** *The node* ∗current *with* extra_entry *and pointer* extra_branch *at position*
         position *are divided into nodes* ∗current *and* ∗right_half *separated by a*
         Record median*.*
  **Uses:** *Methods of* **struct** B_node*, function* push_in*. ∗/*

# Function split_node, Action

```
{
  right_half = new B_node<Record, order>;
  int mid = order/2;   //   The entries from mid on will go to right_half.

  if (position <= mid) { //   First case: extra_entry belongs in left half.
    for (int i = mid; i < order − 1; i++) { //   Move entries to right_half.
      right_half->data[i − mid] = current->data[i];
      right_half->branch[i + 1 − mid] = current->branch[i + 1];
    }
    current->count = mid;
    right_half->count = order − 1 − mid;
    push_in(current, extra_entry, extra_branch, position);
  }

  else {   //   Second case: extra_entry belongs in right half.
    mid++;   //   Temporarily leave the median in left half.
    for (int i = mid; i < order − 1; i++) { //   Move entries to right_half.
      right_half->data[i − mid] = current->data[i];
      right_half->branch[i + 1 − mid] = current->branch[i + 1];
    }
    current->count = mid;
    right_half->count = order − 1 − mid;
    push_in(right_half, extra_entry, extra_branch, position − mid);
  }

  median = current->data[current->count − 1];
              //   Remove median from left half.
  right_half->branch[0] = current->branch[current->count];
  current->count−−;
}
```
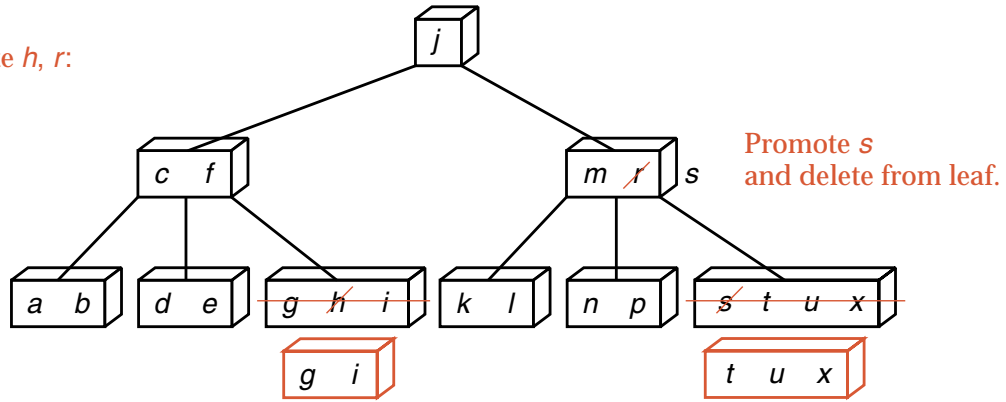
# Deletion from a B-Tree

■ If the entry that is to be deleted is not in a leaf, then its immediate predecessor (or successor) under the natural order of keys is guaranteed to be in a leaf.

■ We promote the immediate predecessor (or successor) into the position occupied by the deleted entry, and delete the entry from the leaf.

■ If the leaf contains more than the minimum number of entries, then one of them can be deleted with no further action.

■ If the leaf contains the minimum number, then we first look at the two leaves (or, in the case of a node on the outside, one leaf) that are immediately adjacent to each other and are children of the same node. If one of these has more than the minimum number of entries, then one of them can be moved into the parent node, and the entry from the parent moved into the leaf where the deletion is occurring.

■ If the adjacent leaf has only the minimum number of entries, then the two leaves and the median entry from the parent can all be combined as one new leaf, which will contain no more than the maximum number of entries allowed.

■ If this step leaves the parent node with too few entries, then the process propagates upward. In the limiting case, the last entry is removed from the root, and then the height of the tree decreases.
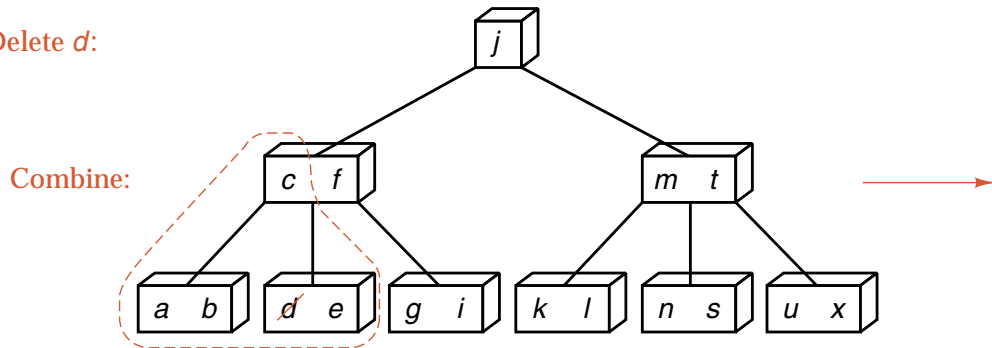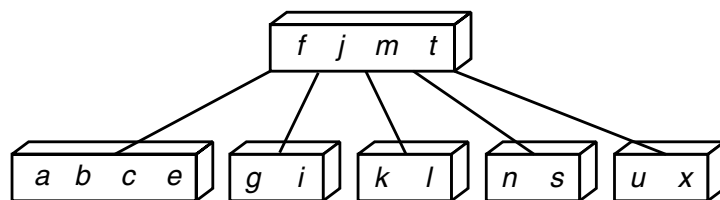
**1. Delete *h*, *r*:**

Promote *s*
and delete from leaf.

**2. Delete *p*:**

Pull *s* down;
pull *t* up.
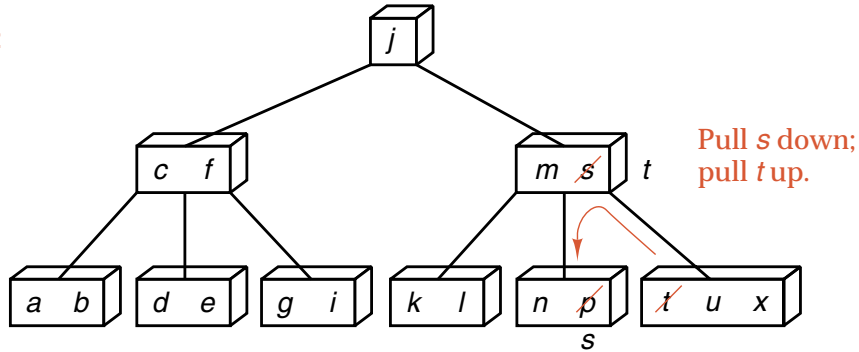
**3. Delete *d*:**

Combine:

Combine:
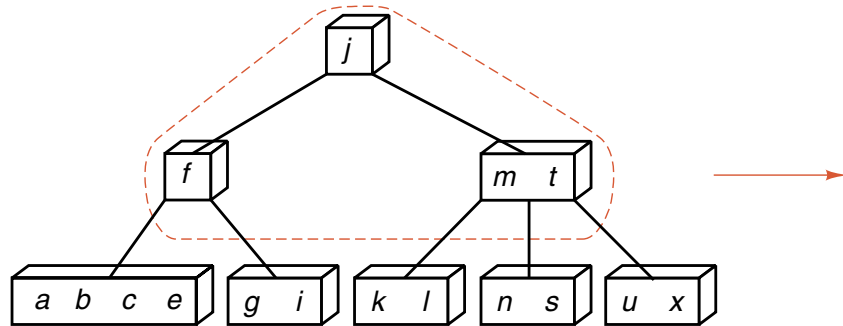
# Public Deletion Method

**template** <**class** Record, **int** order>

Error_code B_tree<Record, order>::remove(**const** Record &target)

/\* **Post:** *If a* Record *with* Key *matching that of* target *belongs to the* B_tree*, a code of*
  success *is returned and the corresponding node is removed from the B-tree.*
  *Otherwise, a code of* not_present *is returned.*

   **Uses:** *Function* recursive_remove \*/

```
{
  Error_code result;
  result = recursive_remove(root, target);
  if (root != NULL && root->count == 0) {  //   root is now empty.
    B_node<Record, order> *old_root = root;
    root = root->branch[0];
    delete old_root;
  }
  return result;
}
```

Recursive Deletion

```
template <class Record, int order>
Error_code B_tree<Record, order>::recursive_remove(
  B_node<Record, order> *current, const Record &target)
```
/* **Pre:**   current *is either* NULL *or points to the root node of a subtree of a* B_tree.
  Post: *If a* Record *with* Key *matching that of* target *belongs to the subtree, a*
       code of success *is returned and the corresponding node is removed from the*
       subtree so that the properties of a B-tree are maintained. Otherwise, a code of
       not_present *is returned.*
  Uses: *Functions* search_node, copy_in_predecessor, recursive_remove *(re-*
       cursively), remove_data, *and* restore. */
```
{ Error_code result;
  int position;
  if (current ==  NULL) result = not_present;
  else {
    if (search_node(current, target, position) ==  success) {
                                  //     The target is in the current node.
      result = success;
      if (current->branch[position] != NULL) { //   not at a leaf node
        copy_in_predecessor(current, position);
        recursive_remove(current->branch[position],
                        current->data[position]);
      }
      else remove_data(current, position);   //   Remove from a leaf node.
    }
    else result = recursive_remove(current->branch[position], target);
    if (current->branch[position] != NULL)
      if (current->branch[position]->count < (order − 1)/2)
        restore(current, position);
  }
  return result;
}
```

Auxiliary Functions

Remove data from a leaf:
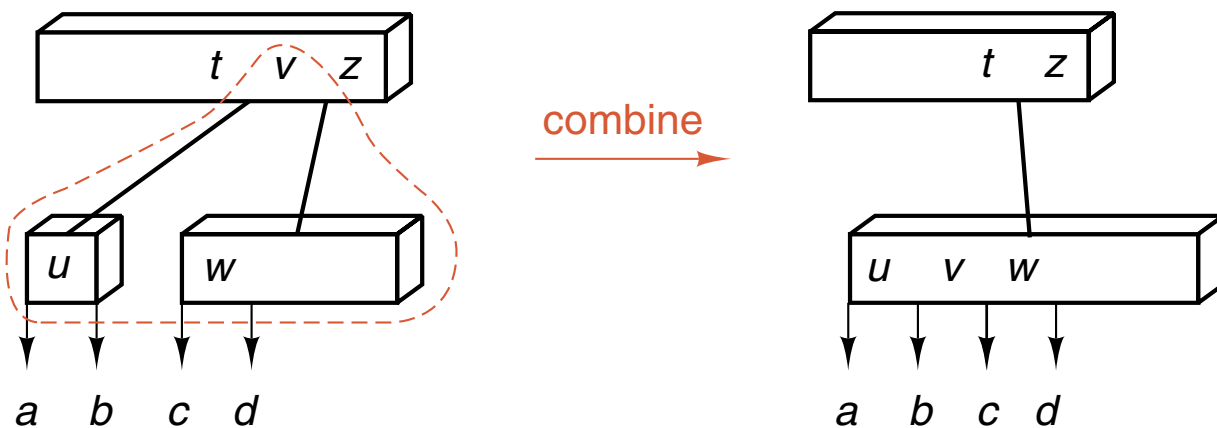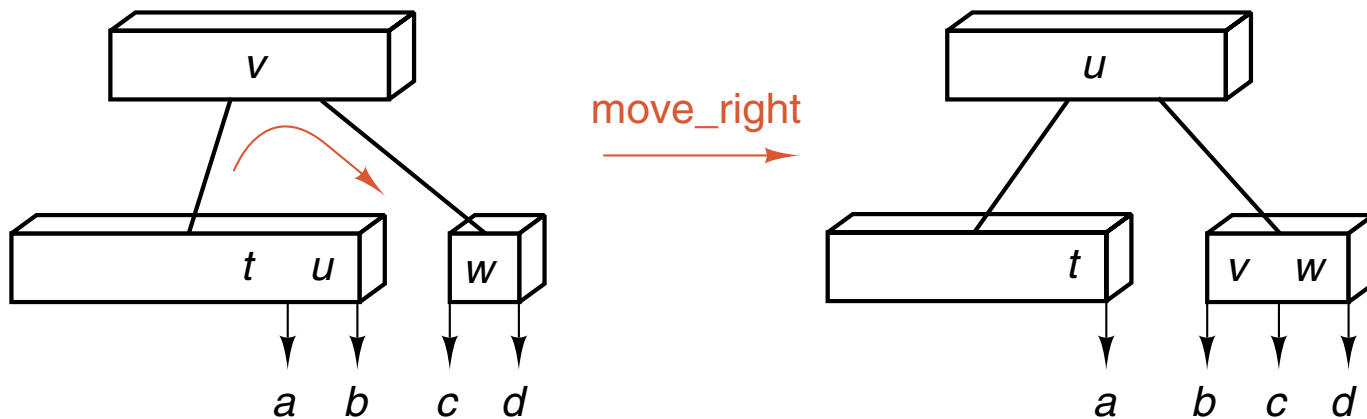
```
template <class Record, int order>
void B_tree<Record, order> ::
            remove_data(B_node<Record, order> *current,
                            int position)
/* Pre:   current points to a leaf node in a B-tree with an entry at position.
   Post:  This entry is removed from *current.  */
{
   for (int i = position;  i < current->count − 1;  i++)
      current->data[i] = current->data[i + 1];
   current->count−−;
}
```

Replace data by its immediate predecessor:

```
template <class Record, int order>
void B_tree < Record, order > :: copy_in_predecessor(
                        B_node<Record, order> *current, int position)
/* Pre:   current points to a non-leaf node in a B-tree with an entry at position.
   Post:  This entry is replaced by its immediate predecessor under order of keys.
          */
{
   B_node<Record, order> *leaf = current->branch[position];
                            //    First go left from the current entry.
   while (leaf->branch[leaf->count]  != NULL)
      leaf = leaf->branch[leaf->count];
                            //    Move as far rightward as possible.
   current->data[position] = leaf->data[leaf->count − 1];
}
```

Restore Minimum Number of Entries

Function to Restore Minimum Node Entries

template <**class** Record, **int** order>
void B_tree<Record, order>**::**
   restore(B_node<Record, order> ∗current, **int** position)
/∗ Pre: current *points to a non-leaf node in a B-tree; the node to which*
        current–>branch[position] *points has one too few entries.*
   Post: *An entry is taken from elsewhere to restore the minimum number of entries in*
        the node to which current–>branch[position] *points.*
   Uses: move_left, move_right, combine. ∗**/**

```
{
  if (position == current->count) //    case: rightmost branch
    if (current->branch[position − 1]->count > (order − 1)/2)
      move_right(current, position − 1);
    else
      combine(current, position);

  else if (position == 0)          //    case: leftmost branch
    if (current->branch[1]->count > (order − 1)/2)
      move_left(current, 1);
    else
      combine(current, 1);

  else                                 //    remaining cases: intermediate branches
    if (current->branch[position − 1]->count > (order − 1)/2)
      move_right(current, position − 1);
    else if (current->branch[position + 1]->count > (order − 1)/2)
      move_left(current, position + 1);
    else combine(current, position);
}
```

Function move_left

template <**class** Record, **int** order>
void B_tree<Record, order>::
   move_left(B_node<Record, order> *current, **int** position)
/* **Pre:** current *points to a node in a B-tree with more than the minimum number of*
      entries in branch position *and one too few entries in branch* position $-$ 1.
  Post: *The leftmost entry from branch* position *has moved into* current, *which has*
      sent an entry into the branch position $-$ 1. */
{
  B_node<Record, order>
    *left_branch = current->branch[position $-$ 1],
    *right_branch = current->branch[position];

  left_branch->data[left_branch->count] =
    current->data[position $-$ 1]; //   *Take entry from the parent.*

  left_branch->branch[++left_branch->count] =
    right_branch->branch[0];

  current->data[position $-$ 1] = right_branch->data[0];
                     //   *Add the right-hand entry to the parent.*
  right_branch->count$--$;

  for (**int** i = 0; i < right_branch->count; i++) {
                    //   *Move right-hand entries to fill the hole.*
    right_branch->data[i] = right_branch->data[i + 1];
    right_branch->branch[i] = right_branch->branch[i + 1];
  }

  right_branch->branch[right_branch->count] =
    right_branch->branch[right_branch->count + 1];
}

Function move_right

template <**class** Record, **int** order>
void B_tree<Record, order> ::
  move_right(B_node<Record, order> ∗current, **int** position)
/∗ **Pre:** current *points to a node in a B-tree with more than the minimum number of*
       entries in branch position *and one too few entries in branch* position + 1.
  Post: *The rightmost entry from branch* position *has moved into* current, *which has*
      sent an entry into the branch position + 1. ∗/

```
{
  B_node<Record, order>
    ∗right_branch = current->branch[position + 1],
    ∗left_branch = current->branch[position];

  right_branch->branch[right_branch->count + 1] =
    right_branch->branch[right_branch->count];

  for (int i = right_branch->count ; i > 0; i−−) {
                              //    Make room for new entry.
    right_branch->data[i] = right_branch->data[i − 1];
    right_branch->branch[i] = right_branch->branch[i − 1];
  }

  right_branch->count++;
  right_branch->data[0] = current->data[position];
                              //    Take entry from parent.
  right_branch->branch[0] =
    left_branch->branch[left_branch->count−−];
  current->data[position] =
    left_branch->data[left_branch->count];
}
```

Function combine

template <**class** Record, **int** order>
void B_tree<Record, order> **::**
  combine(B_node<Record, order> *current, **int** position)
/* Pre: current *points to a node in a B-tree with entries in the branches* position *and*
        position − 1, *with too few to move entries.*
  Post: *The nodes at branches* position − 1 *and* position *have been combined*
        into one node, which also includes the entry formerly in current *at index*
        position − 1. ***/**

```
{
  int i;
  B_node<Record, order>
    *left_branch = current->branch[position − 1],
    *right_branch = current->branch[position];
  left_branch->data[left_branch->count] =
    current->data[position − 1];
  left_branch->branch[++left_branch->count] =
    right_branch->branch[0];
  for (i = 0; i < right_branch->count; i++) {
    left_branch->data[left_branch->count] =
      right_branch->data[i];
    left_branch->branch[++left_branch->count] =
      right_branch->branch[i + 1];
  }
  current->count−−;
  for (i = position − 1; i < current->count; i++) {
    current->data[i] = current->data[i + 1];
    current->branch[i + 1] = current->branch[i + 2];
  }
  delete right_branch;
}
```