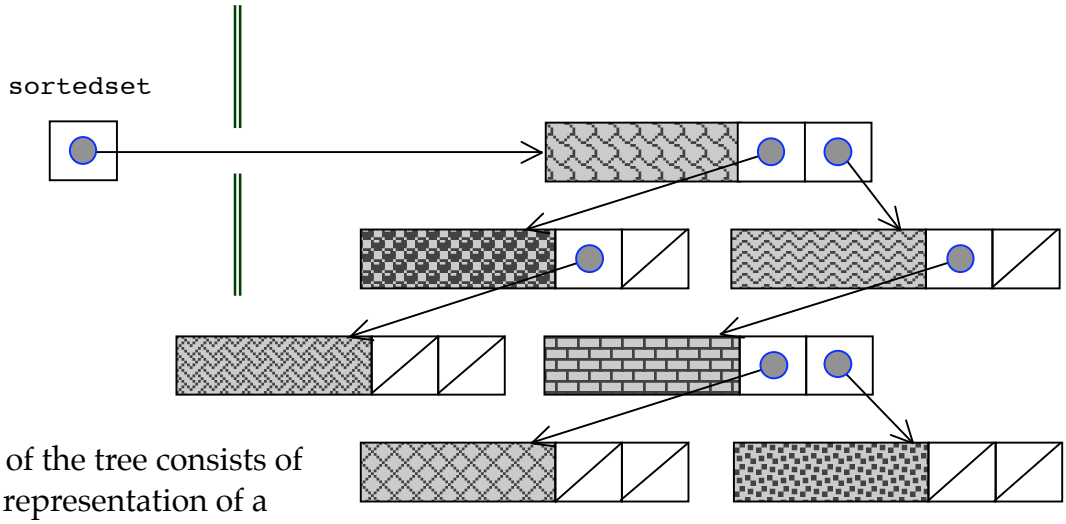
Section Handout

Handout written by Jerry.

The Generic Sorted Set

Binary search trees (BST) save the day whenever both search and insertion are high-priority operations. BST structures have the advantage over both sorted arrays and sorted linked lists in that insertion takes logarithmic time on the average, whereas insertion into sorted lists and arrays generally takes linear time. For this problem, we are going to implement a fully generic **`sortedset`** container using a packed binary search tree as the underlying representation.

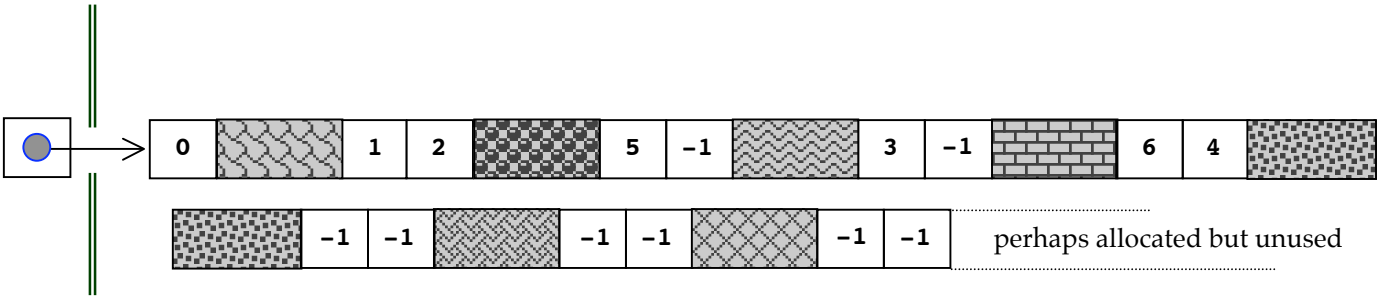
A more traditional binary search tree storing 7 client elements might look as follows:



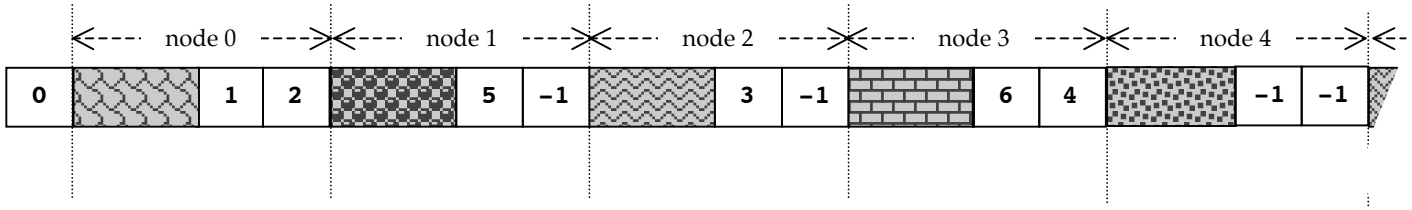
Each node of the tree consists of the binary representation of a client element immediately followed by two addresses, one for the left subtree, and one for the right subtree. Satisfying the binary search tree property mandates that:

be a strictly increasing sequence according to whatever function the client uses to compare elements of this type.

The primary disadvantage here is that the heap gets fragmented with several—potentially hundreds of thousands—of nodes. A more compact, efficient allocation technique packs all of the above nodes into one contiguous block:



Each entry in this packed binary search tree represents a node. Notice that each entry is an element followed by two integers. These two integers replace the two pointers of the traditional BST node. In some ways, they function as links in the same way the pointers do, because they store the indices of the left and right entries that would hang from that element in the normal BST drawing. In the case of node 0, the presence of 1 and then 2 implies that node 1 is the root of the left subtree and node 2 is the root of the right subtree. This pair-of-integers scheme is applied recursively, from root to leaf along every path. The -1 is the integer equivalent of **`NULL`**: To store -1 is to state that no subtree exists.



Notice the one integer to the left of node 0 in the picture. That's more or less the index of the root of the tree. This value is always either -1 or 0, depending on whether or not the BST is empty. We could survive without this extra integer, but we more easily exploit the recursive nature of the data structure when every single node, including the root, can be identified in precisely the same way.

This week's discussion section features the **`sortedset`**. You'll spend the time completing the sorted set declaration and implementation four functions. (Notice that we're not bothering with **`SetFree`** or **`SetRemove`**; that's just too much for one hour.)

```
typedef struct {
   // to be completed
} sortedset;

void SetNew(sortedset *set, int elemSize,
            int (*cmpfn)(const void *, const void *));
bool SetAdd(sortedset *set, const void *elemPtr);
void *SetSearch(sortedset *set, const void *elemPtr);
```

Documentation for each of the three functions is presented below.

```
/*
 * Function: SetNew
 * Usage: SetNew(&stringSet, sizeof(char *), StringPtrCompare);
 *        SetNew(&constellations, sizeof(pointT), DistanceCompare);
 * ----------------
 * SetNew allocates the requisite space needed to manage what
 * will initially be an empty sorted set.  More specifically, the
 * routine allocates space to hold up to 'kInitialCapacity' (currently 4)
 * client elements.
 */

static const int kInitialCapacity = 4;
void SetNew(sortedset *set, int elemSize,
            int (*cmpfn)(const void *, const void *));

/*
 * Function: SetSearch
 * Usage: if (SetSearch(&staffSet, &lecturer) == NULL)
 *            printf("musta been fired");
 * -------------------
 * SetSearch searches for the specified client element according
 * the whatever comparison function was provided at the time the
 * set was created. A pointer to the matching element is returned
 * for successful searches, and NULL is returned to denote failure.
 */

void *SetSearch(sortedset *set, const void *elemPtr);

/*
 * Function: SetAdd
 * Usage: if (!SetAdd(&friendsSet, &name)) free(name);
 * ----------------
 * Adds the specified element to the set if not already present.  If
 * present, the client element is not copied into the set.  true
 * is returned if and only if the element at address elemPtr
 * was copied into the set.
 */

bool SetAdd(sortedset *set, const void *elemPtr);
```