CS107 Handout 16
Autumn 2000 October 12, 2000

The Sorted Set Generic

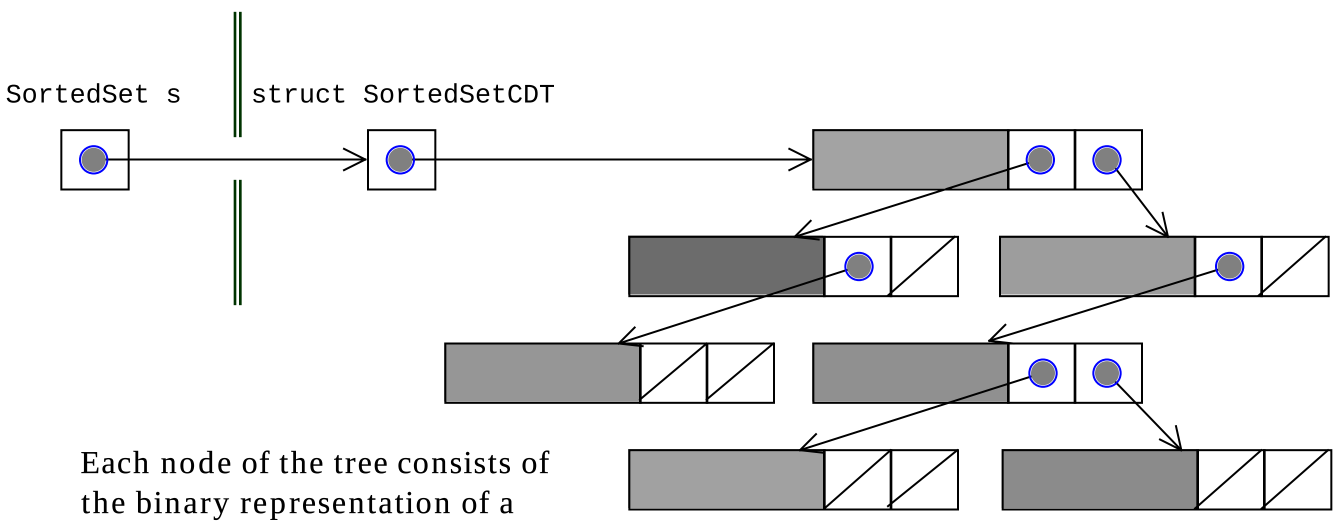
Handout written by Jerry.

This week's larger section problem is derived from the final problem of last spring's CS107 midterm. It is a true generic container type with many of the same features you'll be encountering in the Darray over the course of the next week or so.

The Polymorphic Sorted Set Type

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, you are going to implement a fully generic `SortedSet` ADT using a <u>packed</u> binary search tree as the underlying representation.

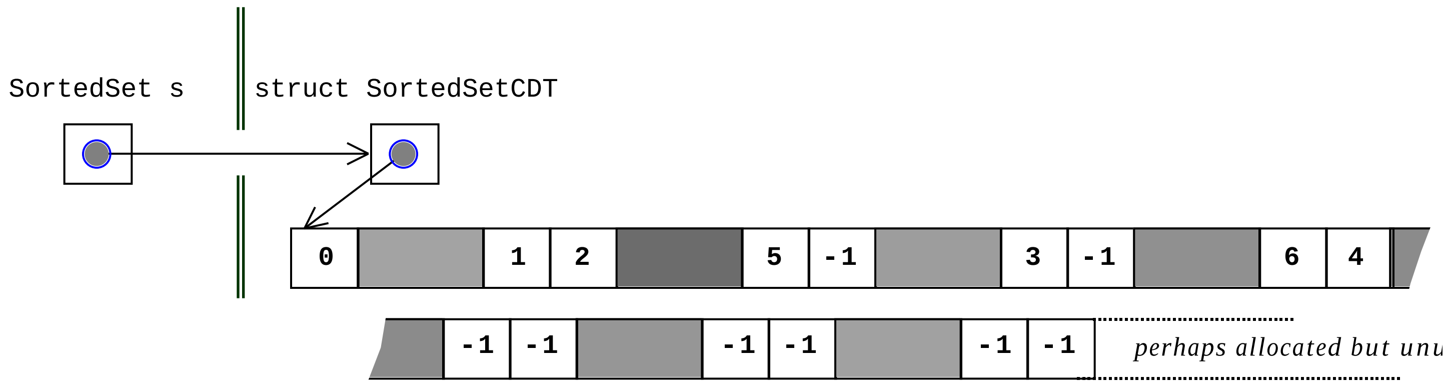
A more traditional binary search tree storing 7 client elements might look as follows:



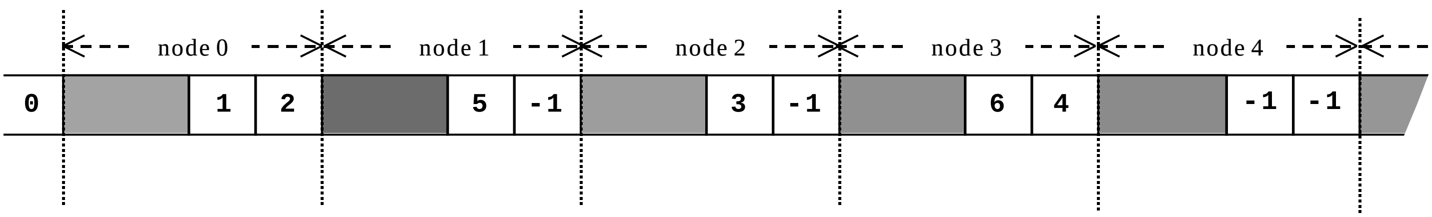
Each node of the tree consists of the binary representation of a client element immediately followed by two addresses, one for the left subtree, and one for the right subtree. Satisfying the binary search tree property mandates that:

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

The primary disadvantage of the above representation stems from the fact that the heap would be fragmented with lots of small nodes. A more compact, efficient allocation technique packs all of the above nodes into one contiguous block, much as the `DArray` in Homework 1b does:



Instead of storing two pointers per node, each node stores two integers which, if treated as indices into an implied array, serve to identify the location of the two nodes of what are logically the left and right subtrees. A sentinel value of `–1` is used to note that the relevant subtree is actually empty. The indices for each node are easily inferred if we conceptually partition the packed nodes as if they were laid out in a true array.



You are going to implement a generic `SortedSet` using this packed binary search tree idea. More specifically, you are going to implement the functionality associated with the following reduced interface:

```
typedef struct SortedSetCDT *SortedSet;

SortedSet SetNew(int elemSize, int (*cmpfn)(const void *, const void *));
bool       SetAdd(SortedSet set, void *elemPtr);
void      *SetSearch(SortedSet set, void *elemPtr);
void      *SetSearchByCriterion(SortedSet set,
                                 bool (*searchfn)(const void *, void *),
                                 void *auxData);
```

A couple of notes before you begin:

- Ensure that you understand the data structure and the manner in which the client elements are packed together.
- Read the header comments very very carefully. They'll provide specific information regarding my expectations, and your implementations must conform to whatever constraints they impose.
- You needn't worry about implementing a `SetFree` function or in any way handling the freeing of client elements. You also needn't be concerned with alignment restrictions.
- You should initially allocate space for 4 client elements, and double the number of allocated elements every time raw storage is saturated.

a. First complete the `SortedSet` type by defining your `struct SortedSetCDT`, and based on your completed type, provide the implementation of the `SetNew` function.

```
struct SortedSetCDT {









};

/*
 * Function: SetNew
 * Usage: dictionary = SetNew(sizeof(char *), StringPtrCompare);
 *        coordinates = SetNew(sizeof(pointT), DistanceCompare);
 * ----------------
 * SetNew allocates the requisite space needed to manage what
 * will initially be an empty sorted set.  More specifically, the
 * routine dynamically allocates space for the concrete type and
 * also allocates space to hold up to 'kInitialCapacity' (currently 4)
 * client elements.  You needn't bother with any calls to assert.
 */

static const int kInitialCapacity = 4;
SortedSet SetNew(int elemSize, int (*cmpfn)(const void *, const void *))
{
```

```
/*
 * Function: SetNew (continued)
 * Usage: dictionary = SetNew(sizeof(char *), StringPtrCompare);
 *        coordinates = SetNew(sizeof(pointT), DistanceCompare);
 * ----------------------------
 */
```

b. A good amount of code is shared by the `SetSearch` and the `SetAdd` functions. This shared code is consolidated to the `FindNode` function, which should be implemented to conform to the following specification. You have this and the next page for your implementation.

```
/*
 * Function: FindNode
 * Usage: ip = FindNode(set, elem);
 *        if (*ip == -1) printf("indexp points where this element belongs!");
 * ------------------
 * FindNode descends through the underlying binary search tree of the
 * specified set and returns the address of the offset into raw storage
 * where the specified element resides.  If the specified element isn't
 * in the set, FindNode returns the address of the –1 that would be updated
 * to contain the index of the element being sought if it were the
 * next element to be inserted——that is, the address of the –1 that ended
 * the search.  You needn't bother with assert.
 */

static int *FindNode(SortedSet set, void *elem)
{
```

c. Using your `FindNode` routine, provide implementations for `SetSearch` and `SetAdd`.

```
/*
 * Function: SetSearch
 * Usage: if (SetSearch(staff, &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.  Because the binary search tree property is enforced
 * according to this comparison function, FindNode does most of your work
 * for you.  A pointer to the matching element is returned for successful
 * searches, and NULL is returned to denote failure.
 */

void *SetSearch(SortedSet set, void *elem)
{
```

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

bool SetAdd(SortedSet set, void *src)
{
```

d. Finally, implement the more flexible `SearchSetByCriterion`, which allows the client to specify a two argument predicate function and auxiliary data that can used as the second argument. Note that the binary sreach tree property is irrelevant here, and that the search may take linear time. The full specification is provided below:

```
/*
 * Function: SetSearchByCriterion
 * Usage: match = SetSearchByCriterion(dictionary,
 *                                     ContainsOnlyTheseCharacters, "abcdefg");
 *        match = SetSearchByCriterion(coordinates,
 *                                     CloseEnoughTo, sanFranciscoCoordinate);
 * ------------------------------
 * SetSearchByCriterion searches for the first client element
 * for which the specified predicate function returns true.
 * The predicate function takes a pointer to a client element
 * as its first argument, and takes the auxData variable as its
 * second.  If more than one client element passes the specified
 * predicate test, then any one might be returned.  If the search
 * is successful, the a pointer to a matching client element is
 * returned.  Otherwise, NULL is returned to denote failure.
 */

void *SetSearchByCriterion(SortedSet set,
                           bool (*searchfn)(const void *elem, void *auxData),
                           void *auxData)
{
```

```
struct SortedSetCDT {
   int *root;          // points to the offset index of the root.
   int logicalSize;    // number of active client elements currently stored.

   int allocatedSize;  // number of elements to saturate allocated memory.
   int (*cmp)(const void *, const void *);
   int elemSize;       // client element size.
};

#define   NodeSize(clientElem)  ((clientElem) + 2 * sizeof(int))

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

   assert(elemSize > 0);
   assert(cmpfn != NULL);

   set = malloc(sizeof(struct SortedSetCDT));
   assert(set != NULL);

   set->root = malloc(sizeof(int) + kInitialCapacity * NodeSize(elemSize));
   assert(set->root);

   *set->root = -1;        // set it empty
   set->logicalSize = 0;   // still empty
   set->allocatedSize = kInitialCapacity;
   set->cmp = cmpfn;
   set->elemSize = elemSize;
   return set;
}

static int *FindNode(SortedSet set, void *elem)
{
   void *curr;
   int comp, *root = set->root;

   root = set->root;
   while (*root != -1) { // while not addressing a leaf
      curr = (char *) (set->root + 1) + *root * NodeSize(set->elemSize);
      cmp = set->cmp(elem, curr);  // compare client element to value at curr
      if (cmp == 0) break;
      root = (int *)((char *) curr + set->elemSize);
      if (cmp > 0) root++;
   }

   return root;
}
```

```
void *SetSearch(SortedSet set, void *elem)
{
   int *node = FindNode(set, elem);
   if (*node == -1) return NULL;
   return (char *) (set->root + 1) + *node * NodeSize(set->elemSize);
}

static void Expand(SortedSet set)
{
   set->allocatedSize *= 2;
   set->root = realloc(set->root,
                  sizeof(int) + set->allocatedSize * NodeSize(set->elemSize));
}

bool SetAdd(SortedSet set, void *src)
{
   int *child;
   void *dest;

   child = FindNode(set, src);
   if (*child != -1) return false;  // already there.. say we didn't add it.

   if (set->logicalSize == set->allocatedSize) Expand(set);
   *child = set->logicalSize++;

   dest = (char *) (set->root + 1)+ (*child) * NodeSize(set->elemSize);
   memcpy(dest, src, set->elemSize);
   child = (int *)((char *) dest + set->elemSize);
   *child++ = -1;
   *child = -1;
   return true;
}

void *SetSearchByCriterion(SortedSet set,
                           bool (*searchfn)(const void *elem, void *auxData),
                           void *auxData)
{
   int i;
   void *curr;

   for (i = 0; i < set->logicalSize; i++) {
      curr = (char *) (set->root + 1) + i * NodeSize(set->elemSize);
      if (searchfn(curr, auxData)) return curr;
   }

   return NULL;
}
```