Problem 3-1. Greedy Crossword Grid

Consider a general crossword grid: an $n \times n$ grid with some squares black and some squares white. A maximal vertical or horizontal sequence of white squares (where maximal means it is terminated on each end by a black square or the edge of the grid) is called a “wordspace”.

Suppose you want to find the minimal number of wordspaces that contain all of the white squares.

(a) Show that the greedy algorithm of successively “picking a wordspace including the most white squares not yet included” doesn’t give an optimal solution.

Solution: Consider the following diagram and assume that every square not shown is shaded black:

```
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
```
Using the greedy approach mentioned, we would choose the horizontal strip of four squares and then would choose each of the remaining single squares for a total of five. However, the optimal approach would be to take each of the four vertical strips.

(b) Give a simple approximation algorithm that always returns a solution (a collection of wordspaces including all of the whitespace) that has not more than twice the minimum possible number of wordspaces in a solution.

**Solution:**

1. Pick any white square that is not already covered.
2. Take both the horizontal and vertical wordspaces that intersect at that square.
3. Repeat until every white square is covered.

**Proof:**

- Whenever you pick a square, it’s not covered by any of the wordspaces you’ve chosen so far. So you know that no two of the squares picked in step 1 are in the same wordspace.
- Think about the set of squares we end up choosing in step 1 of the algorithm. Let’s say there are $k$ of them. Any arrangement of wordspaces that covers the whole grid obviously has to cover all $k$ of those squares. But we know that no two of them are part of the same wordspace, so any arrangement that covers these squares must contain at least $k$ wordspaces.
- But we’ve covered the grid using $2k$ wordspaces. So, we have a 2-approximation.

**Problem 3-2. Deterministic Skip List**

The skip list implementation shown in class is a quick and easy way to get a data structure with good expected runtime. However, the worst case behavior is awful. By adding limitations to how high an element can be promoted, we can safeguard against truly bad performance. In this problem, we’ll see that we can go a step further and make the skip list fully deterministic.

First, let’s define the notion of promoting an element through height $k$ to mean that it exists at all heights from 1 to $k$.

The key property to a deterministic skip list is that between two adjacent elements of height $h$ with $h \geq 2$, there are a minimum of $B - 1$ and a maximum of $2B - 1$ elements of height $h - 1$. For this problem, feel free to assume that $B = 2$. For clarity, elements in the skip list start out at height 1 and are promoted up through some height equal to or greater than 1. We also specify that right before the first element and right after the last element are elements which are always promoted through one more level than the current height of the rest of the skip list and have values of $-\infty$ and $+\infty$, respectively. See figure 1 for an example.

In this problem, $n$ is the number of elements in the skip list.
Figure 1: Notice how before and after the elements in our skip list are $\pm\infty$ which are elements promoted one higher than the rest. Also notice that the key property is satisfied. There are two elements at height 3 and between them are two elements of height 2 (the key property dictates that there can be one, two, or three at height 2 between them). There are four elements at height 2 since the $\pm\infty$ are also at height 2 by definition. Thus, there are between one and three elements between each adjacent pair of them (i.e. there’s one between $-\infty$ and 2, one between 2 and 5, and two between 5 and $\infty$).

(a) Show that the height of our skip list is $O(\log n)$.

Solution: Let $h$ be the height of the skip list. To find the maximum value for $h$, we’ll want there to be 1 element between every two elements of height $k$ at height $k - 1$ since if we had 2 or 3, that could only cause the height to decrease. Thus, if we have one at height $h$, we must have two at height $h - 1$ since we have nodes before and after the skip list that are always promoted through one more than the height. At height $h - 2$, we’ll then have 4 elements. Continuing in this fashion, we see that the number of elements at each height doubles as we go down giving us this equation:

$$n = \sum_{i=1}^{h} 2^i.$$ 

Evaluating that sum gives

$$n = 2^h - 1 \Rightarrow h = O(\log n)$$

as desired.

(b) Prove that performing a SEARCH operation takes $O(\log n)$ time.

Solution: The key property to realize is that at each level we only have to go over at most 3 elements before we are sure we will go down a level. Thus, since there are
$O(\log n)$ levels, and we do $O(1)$ comparisons at each level, searching takes $O(\log n)$.

(c) Explain how to implement the `INSERT` operation such that the key property is always maintained and the operation still runs in $O(\log n)$ time.

**Solution:** We first find where the element will be inserted by searching the skip list as usual. Then we insert it at this location at height 1. If this causes the key property to be violated meaning that there are now four elements of height 1 between two of height 2, then we can fix this by promoting the second of these elements. Again, we may violate the key property, so we continue to fix it by promoting the second element all the way up, if necessary.

The search takes $O(\log n)$ time and fixing the key property will take time proportional to the height of the tree which will be $O(\log n)$.

(d) The property that between any two elements of height $h > 1$, there are between $B - 1$ and $2B - 1$ elements of height $h - 1$ should remind you of another data structure we’ve looked at in this class. Which data structure is that? Just name the data structure and give a short 1 or 2 sentence explanation.

**Solution:** B-Trees. In a B-Tree we have that each node has between $t - 1$ and $2t - 1$ keys.
Problem 3-3. B-Trees and Paging to Disk

In data-intensive applications, disk transfers can account for the bulk of the running time of an algorithm—much more so than any calculations that are performed on the data. As described in CLRS (Ch. 18), B-trees are often used in real-world applications in order to optimize the number of disk transfers.

When laying out data structures on disk, we often have several options. While a simple structure like an array is typically just a block of contiguous, indexed memory, more complicated structures like lists and trees make use of pointers—thus, while the conceptual layout of the structure may be a list or tree, the physical location of each node on disk can in principle be arbitrary. However, many operations on these structures occur in predictable patterns, so it is beneficial to organize the physical layout of the data structure in such a way as to be efficient as possible. For this problem, we will be exploring methods of laying out the B-tree data structure on disk so as to minimize the cost of a search.

For this problem, assume that:

- Computations can only be performed on data that is in memory (rather than on disk).
- The disk is broken up into equal-sized regions of contiguous memory called pages. Data may be read from disk to memory, or written from memory to disk, only an entire page at a time. Reading or writing a single page takes \( \Theta(1) \) time. Pages can be accessed in a random-access fashion (don’t worry about seeking to a page).
- Pages are automatically read to memory when following a pointer to data on that page.
- Computations performed on data in memory are instantaneous; they do not affect the running time.
- The memory is initially empty, and can hold \( \Theta(1) \) pages. The disk initially holds the B-tree.
- The B-tree contains a total of \( n \) elements and has branching factor \( t \geq 2 \). Denote the size of a page (in bytes) as \( P \) and the maximum size of a node (in bytes, including all relevant values and pointers) as \( y_t \). (Note that since a node stores \( \Theta(t) \) values, \( y_t \) is a function of \( t \)).

(a) Assume that \( P = y_t \); that is, that one B-tree node exactly fits into a page. How would you arrange the B-tree nodes on disk? What is the running time for a single search operation?

Solution: Put each node in a separate page. As you go down the B-tree, each level results in a random-access page read, which costs \( \Theta(1) \). Thus, the time taken is \( \Theta(\log_t n) \).
(b) Assume that $P = y_t$ as in part (a), but that you do not know the point at which one page ends and the next page begins on disk. (You know, however, that each page is contiguous.) If you use the same design as in your solution to part (a), what is the asymptotic running time of a single search?

**Solution:** Even if we cross a page boundary, each node will simply be on two consecutive pages instead of a single page. Thus, we will end up loading twice the number of pages, but the asymptotic performance is still $\Theta(\log t n)$.

(c) Assume that $P = y_t \cdot (1 + t + t^2 + \ldots + t^{k-1})$ for some known value $k$. Describe how you might arrange the B-tree nodes on disk to minimize the number of accesses. Analyze the running time for a single search. (Your running time is allowed to depend on $k$.)

**Solution:** Divide the B-tree up into height-$k$ subtrees. The first level has one (the first $k$ levels of the tree itself), the second level has $t^k$ independent subtrees, and so on. Store each of these subtrees in a page. Each time we reach the bottom of one of the subtrees in our B-tree search, we jump to the top of another; each search accesses only one subtree at each level. Thus, a search takes $\Theta((\log t n)/k)$ time.

(d) Now assume that you’re writing software that can run on a wide range of hardware. Thus, there could be many possible page sizes $P$; however, you are allowed to assume that $P \geq y_t$ (that is, a page is at least as large as a single B-tree node). Describe a method to lay out the data structure on disk in order to minimize the number of disk accesses no matter what the value of $P$ is. Analyze the running time of a single search (your answer is allowed to depend on $P$ and/or $y_t$). If it happens that $P = y_t \cdot (1 + t + t^2 + \ldots + t^{k-1})$, how does the asymptotic performance of this layout compare with the solution you gave to part (c) (which was designed with knowledge of $P$)?

*Hint:* Examine what happens in part (c) in the specific case where $k = (\log_t n)/2$, and apply the technique recursively.

**Solution:** First, break the tree in half by height; this results in $\sqrt{n} + 1$ subtrees of height $(\log_t n)/2$ (see Fig. 2). Lay these out in arbitrary order—but recurse on each subtree, dividing its height in half and laying out those subtrees in some order, and so forth, until our subtrees consist of single nodes. Note that the subtree of height $x$ will contain $1 + t + t^2 + \ldots + t^x = \Theta(t^x)$ nodes, and thus take up $\Theta(t^x y_t)$ bytes.

If $P$ exactly corresponds to the size of one of these subtrees (say, the one at height $k$), we know that $P = \Theta(t^k y_t)$. By (c) the running time is $\Theta((\log_t n)/k) = \Theta(\frac{\log_t n}{\log_t (P/y_t)}) = \Theta(\log_P y_t n)$.
Figure 2: Successively breaking the tree in half by height. To lay out each subtree, take its subtrees and lay them out successively.

If $P$ does not correspond to the size of one of the subtrees, it falls between two sizes. But this means that there is some subtree of height $i$ such that $t^i < P/yt < t^{2i}$. And this $i$ is still therefore $\Theta(\log_t(P/yt))$. Note that each subtree of height $i$ is contiguous, and therefore requires at most two page reads (in the case that it is aligned across a page boundary, as in part b). The total number of page reads to search down the B-tree is therefore still only $\Theta(\log(P/yt) n) = \Theta(\log(P/yt) n)$.

Thus, we get the same asymptotic performance for this layout as part (c) even without knowing $P$.

**Problem 3-4. Lazy BSTs**

Ben Bitdiddle picked up some bad habits as an MIT student. Specifically, he tended to wait until the last possible moment to do his problem sets, and when he finally did them, he only did a good enough job to get by. Unfortunately, he’s continued this approach to problem-solving: when asked to build a binary search tree supporting inserts and searches, he came up with a design that only partially built the tree.

Ben’s tree works like this:

- The basic design is a binary tree whose internal nodes are single elements as usual, but whose leaves are arrays.

- A leaf at depth $k$ is an array of size $2^k$. (The root has depth 0.)

---

2Don’t be like Ben. His friend Alyssa is a much better role model.
- Inserting an element works like normal in a binary search tree until one reaches the leaves. At that point, Ben is lazy enough that he decides he’s had enough of this tree stuff, and just tacks the element on the end of the (unsorted) array if there’s room.

- If the array is full on an insert, Ben sighs, finds the median of the array (including the new element) using the linear-time algorithm seen in class, and partitions the array around it by value. He therefore has the median element and two unsorted subarrays: one with all elements of the original array less than the median, and one with all elements of the original array greater than the median. He sets the median as an internal node and sets the two subarrays as its two children, turning the former leaf into an internal node and making two new leaves.

- To search, one performs a normal BST search until one reaches a leaf, then looks through all of the array elements to see if it is there.

![Figure 3: Ben’s lazy BST design.](image)

(a) Assume that the tree contains $n$ elements. While it is not necessarily perfectly balanced, it has been constructed by starting with an empty tree and applying $n$ insert operations as described above. What is the worst-case running time of an additional insert operation that does not cause the array to overflow?

**Solution:** The worst case is when the tree is as imbalanced as it can be, and is therefore similar to a single chain. However, note that when we create new children for a node, we split the previous array at that node equally. Thus, the chain must have “side” children of size $0, 1, 2, 4...$ elements (see Fig. 4).

Assume the tree is of height $k$. Then the total number of elements in the tree is $0 + 1 + 2 + ... + 2^{k-1}$ for the side children, plus $k$ for the internal nodes, plus $2^{k-1}$ for the right child, for a total of $2^k + 2^{k-1} + k - 1$. Thus, $k = \Theta(\log n)$. Since inserting an element onto the end of the array is $\Theta(1)$, the running time is $\Theta(\log n)$.

(b) Assume again that the tree has been built up using $n$ insert operations. What is the worst-case cost of an insert (that does cause the array to overflow)? What is the cost per operation using amortized analysis?
Figure 4: Even the most unbalanced tree must have some elements to the side of the main chain.

**Solution:** The insertion costs $\Theta(\log n)$ to traverse the tree, followed by $\Theta(2^k) = \Theta(n)$ to partition the array (since as seen in class, we can do this in linear time in the size of the array). This results in a worst-case cost of $\Theta(n)$.

For the amortized cost, we know that the actual cost is $\Theta(m)$ for an array of size $m$; let $c$ be a constant such that the work done is at most $cm$. Charge an additional $2c$ units of work for each insertion. The additional $\Theta(1)$ work clearly does not affect the asymptotic bound in the case where there is no array overflow.

If there is an array overflow at level $k$, note that the array started with size $2^{k-2}$ (half of the maximum size of the parent’s array), and is now of size $2^k$. Thus, $3 \cdot 2^{k-2}$ insert operations must have put elements into this array. We draw on that $3c \cdot 2^{k-1} > c \cdot 2^k$ stored work to partition our array at no extra amortized cost. Thus, the amortized cost per operation is simply $\Theta(\log n)$.

This can also be phrased as a potential argument. Let $c$ be some constant to be determined later. Define a potential function $\Phi$ as $\Phi = c \cdot (2n - \sum_{x \text{ is not a leaf}} 2^{\text{depth}(x)})$.

The initial empty tree has potential 0, and the potential is always nonnegative (since an internal node at depth $k$ will always have at least $2^k - 2^{k-2} > 2^{k-1}$ descendents due solely to insertions made at that node, so these descendents will outweigh the $2^k$ weight of this internal node since our potential function doubles $n$). Each non-splitting insert does not change the number of internal nodes, so it simply adds $2c$ to the potential (since $n$ increases by 1). Splitting a node at depth $k$ takes $\Theta(2^k) \leq c \cdot 2^k$ work, which is paid for by the decrease in potential corresponding to the new internal node (assuming we let $c$ be the constant factor corresponding to the time to partition the array).

Other potential functions could possibly work as well.

(c) Assume that the tree is initially empty, and we know that the pattern of accesses will be $n$ inserts followed by a single search. Perform an amortized analysis of the cost per operation of this access sequence.
Solution: Since there is only a single search operation, even though it could take up to $\Theta(n)$ actual time, we can amortize the cost over the previous inserts by adding $\Theta(1)$ additional potential with each insert operation.

(d) Perform an amortized analysis assuming an arbitrary sequence of inserts and searches. How well does Ben’s approach fare if there are a lot of searches?

Solution: Since an arbitrary number of searches can occur for a fixed number of inserts, there is no way to amortize the cost of looking through the array on each search. Thus, the amortized cost of a search at level $k$ is $\Theta(2^k)$, which is $O(n)$ (and reaches $\Theta(n)$ for a worst-case tree as described in part (a)).