Out-of-core SSet implementations

When the data set is too large to store in main memory, secondary storage must be used, eg. hard drive storage.

Since access time to disk is much slower, this changes the considerations for an SSet implementation.

Search trees work, but major changes are appropriate. You don't want to cause a disk read for every item accessed during a find operation...

b-trees

Key properties of b-trees:

1. 1.1 A b-tree node has from b to 2b children and contains from b-1 to 2b-1 keys.
   1.2 Exception: the root node may have fewer, containing from 2 to 2b children and from 1 to 2b-1 keys.
2. All leaves of a b-tree are at the same depth.
3. A b-tree is a search tree. If a node contains child links $c_0, c_1, \ldots, c_n$ and keys $k_0 < k_1 < \ldots < k_{n-1}$, then we have
   3.1 $k < k_0$, for all keys k in the subtree rooted at c_0,
   3.2 for $1 \leq i \leq n-1$, we have $k_{i-1} < k < k_i$, for all keys k in the subtree rooted at c_i, and
   3.3 $k_{n-1} < k$, for all keys k in the subtree rooted at c_n.

A 2-3-4-tree is the case b = 2 of a b-tree.

HDD (Hard Disk Drive) storage medium

- Hard drive capacities: terabyte becoming common.
- Hard drives may rotate 10,000 rpm.
- Seek time (move head to proper ring) circa 10ms.
- Latency (average time for proper point of disk to rotate under head) circa .3ms.
- A logical block is read at a time
  The old logical block size standard was $512 = 2^9$ bytes, new standard is $4096 = 2^{12}$ bytes.
- 2^{12} bytes $= 2^9$ 8-byte words $= 2$ arrays (of $2b$ words and of 2b-2 words) for $b \approx 2^7$.

So a possibly good value for b is 128.

min and max number of keys in a b-tree of given height

height	*min*	*max*
-1	0	0
0	1	$2b-1$
1	$1+2(b-1)$	$2b-1+2b(2b-1)$
2	$1+2(b-1)+2b(b-1)$	$2b-1+2b(2b-1)+4b^2(2b-1)$
3	$1+(2+2b+2b^2)(b-1)$	$(1+2b+4b^2+8b^3)(2b-1)$
3	$2b^3-1$	$(2b)^4-1)$
$\vdots$	$\vdots$	$\vdots$
h	$O(b^h)$	$O((2b)^{h+1})$

Also, the max at height h is about $b2^h$ times the min at height h.

min and max number of keys in a b-tree of given height

	$b = 2$		$b = 3$		$b = 128$	
ht	$min + 1$	$max + 1$	$min + 1$	$max + 1$	$min + 1$	$max + 1$
-1	1	1	1	1	1	1
0	2	4	2	6	2	256
1	4	16	6	36	256	65536
2	8	64	18	216	32768	16777216
3	16	256	54	1296	4194304	4294967296
4	32	1024	162	7776	536870912	1099511627776
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
h	2^{h+1}	4^{h+1}	$2 \cdot 3^h$	6^{h+1}	$2b^h$	$(2b)^{h+1}$
h	m	$2^{h+1}m$	m	$3 \cdot 2^h$	m	$(b2^h)m$

Flash memory vs Hard Drives

SSD (Solid State Drive) technology competes with HDD.

Pro HDD: capacity, price.
Pro SSD speed, power consumption, mechanical durability.

Flash Drive access time is circa 0.1ms. Compare with vs circa 10ms for HDD. It is at least 100 times faster than the competing HDD under consideration.
There are debates about relative reliability of SSD and HDD. Because of larger capacity HDD is more likely to be in use for an out-of-core SSet application (b-tree application)..

b-tree code sketch

```
/* DRAFT pseudocode sketch of b_Tree.h, containing
the b-tree implementation of SSet operations.
*/

template <class T> struct INPair { // item-node pair
  T item;
  Node* child; //
  INPair(Node* n, T i) { item = i; node = n; }
}

/* a basic node that could work for B-tree */
template <class T>
struct Node {
    INPair<T>* a;
    int k; // current size.
    Node() { a= new INPair<T>[2*b], n = 0; }
    INPair& operator[](int i) { return a[i]; }
};
```

b-tree applications and variants

b-trees work well with the memory hierarchy of fast RAM memory and large, permanent, but slow access hard drive memory, and fairly large b (such as $100 < b < 1000$).
To reduce cost of insertions and deletions, A single node could itself be a balanced binary tree such as a 2-3-4-Tree or Treap, but with all the nodes allocated within the node's block of memory. But this is often not done because the speedup is negligible relative to the cost of the disk reads.
There are many variants,

- a) some aimed at insuring a large root node: vary the min and max number of items per node, for instance put only items, no child pointers in leaf nodes.
- b) maximize keys per node and hence minimize depth by storing all items in leaves only. In this variant, keys in internal nodes are copies of the keys only out of the items in leaves.

b-tree clicker question 1

In a b-tree what is the minimal number of items in the root node?

A. 1

B. b-1

C. 2b-1

D. 2n

b-tree clicker question 2

In a b-tree what is the minimal number of items in a non-root node?

A. 1

B. b-1

C. 2b-1

D. 2n

b-tree clicker question 3

In a b-tree what is the maximal number of items in any node?

A. 1

B. b-1

C. 2b-1

D. 2n

b-tree activity

Let A be a b-tree of characters, with b = 3. Nodes contain a maximum of 6 child links and 5 keys.

A. Determine the b-tree that results from adding a thru e in order.

B. Determine the b-tree that results from adding a thru f in order.

C. Determine the b-tree that results from adding a thru h in order.

D. Determine the b-tree that results from adding a thru i in order.

E. Determine the b-tree that results from adding a thru z in order. What is it's height?

F. Is there an order of insertion of 26 items that would result in a height 1 tree? (b = 3, still.) If so, find such an order.

b-tree clicker question 4

As a function of b, a b-tree of height 2 has how about many *nodes* at minimum? (choose closest answer)

A. 1

B. b

C. b^2

D. b^3

b-tree clicker question 5

As a function of b, a b-tree of height 2 has how about many *nodes* at maximum? (choose closest answer)

A. 1

B. b

C. b^2

D. b^3