Lecture Notes 05
CSE 350 Spring 2026

B-Tree Nodes. A B-Tree can have two types of nodes:

- **Internal nodes** which stores up to B_{max} links (e.g., pointers, page numbers, etc.) to child nodes and up to $B_{max} - 1$ pivot keys (aka separator keys).
- **Leaf nodes** which stores up to B'_{max} data entries (e.g., key-value pairs).

Typically, we want an external-memory B-tree node to span the entire one or multiple page. For specific types of key values, assuming they are both fixed-length, this puts an restriction of how large B_{max} and B'_{max} can be.

If we denote the node size as P, key size as K, link size as C, value size as V, the number of links we store on an internal page as B, the number of pairs we store on a leaf page as B', assuming there is no page header overhead and there is no alignment requirements, we can derive B_{max} and B'_{max} as follows:

$$BC + (B-1)K \leq P \Rightarrow B \leq \frac{P+K}{K+C} \Rightarrow B_{max} = \lfloor \frac{P+K}{K+C} \rfloor$$
$$B'(K+V) \leq P \Rightarrow B' \leq \frac{P}{K+V} \Rightarrow B'_{max} = \lfloor \frac{P}{K+V} \rfloor$$

B-Tree Invariants. A B-Tree has the following invariants:

1. A root page that is also an internal page must have at least 2 links.
2. A non-root internal page must have at least cB_{max} links, where c is an arbitrary constant that satisfies $2/B_{max} \leq c \leq 1/2$. The usual choice is $c = 1/2$, which results in the lowest tree height and search cost.
3. A non-root leaf page must have at lesat $c'B'_{max}$ pairs, where, again, c' is an arbitrary constant that satisfies $2/B'_{max} \leq c \leq 1/2$. The usual choice is $c' = 1/2$, which results in the lowest tree height and search cost.
4. Suppose we denote the internal page as an interleaving of links and keys

   $$c_0, k_1, c_1, k_2, c_2, \ldots, k_{B-1}, c_{B-1}$$

   And all keys strictly increases as index increases ($k_1 < k_2 < \ldots k_{B-1}$), and $\forall i \in [1, B-1], k_i$ is the pivot key that separates the key space between two sub-trees c_{i-1} and c_i. For convenience, we denote $k_0 = -\infty$ and $k_B = +\infty$.
   Then $\forall i \in [0, B-1]$, the key space of c_i, i.e., the range of keys of all data entries in the sub-tree rooted at c_i must be in the range of $[k_{i-1}, k_i)$. This also implies all pivot keys in the sub-tree c_i must also be in the above range. This invariant works for both root and non-root internal pages.

Type alignment. A type T can have an alignment requirement of A, where A is typically power of 2. This often comes up on architectures where unaligned memory reads or writes can have atomicity/performance impliciation (e.g., x86-64), or is disallowed (e.g., certain ARM).

More formally, a type-T variable x satfisies the alignment requirement of A if and only if `&x % A == 0` (where `&x` denotes the address of x). While A does not have a direct correlation with the length of type T (or T could be variable-length), for a fixed-length primitive type such as signed/unsigned integer and floating point number typically have alignment equal to its length.

Type length and alignment impact how a C/C++ compiler computes the layout of a structure. Taking an intuitive B-tree intenral node definition in C++ (with g++ x86-64) as an example. Suppose we have a node size of 4096 bytes, and the key, link and value sizes/alignments are all 4, and we compute B_{max} and B'_{max} using the formula in the first section. The following struct will have a size of exactly 4096 bytes.

```
constexpr size_t Bmax = (4096 + 4) / (4 + 4); // 512
struct BTreeInternalNode {
    int32_t cnt;
    int32_t key[Bmax - 1];
    int32_t links[Bmax];
};
```

However, if the key size/alignment is 8 instead of 4, a similar definition will result in a different struct size 4104, slightly exceeding the page size 4096.

```
constexpr size_t Bmax = (4096 + 8) / (8 + 4); // 342
struct BTreeInternalNode {
    int32_t cnt;
    // 4 bytes hidden padding here
    int64_t key[Bmax - 1]; // starts from offset 8 due to alignment of 8
    int32_t links[Bmax];
};
```

So, if you use C++ struct to define a data structure that you need a precise control on the layout or node size, you have to take alignment into account. Alternatively, you can compute the layout by yourself and work directly on a byte buffer. You will practice both in Project 2, Checkpoints 0 and 1.